GRAPH-BASED RULE AND TRANSACTION EXECUTION IN A PARALLEL ACTIVE OBJECT ORIENTED KNOWLEDGE BASE MANAGEMENT SYSTEM

By

RAMAMOHANRAO SRI JAWADI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1994

ACKNOWLEDGMENTS

I am deeply indebted to Dr. Stanley Y.W. Su, chairman of my supervisory committee, for his continuous guidance, help, inspiring and thought-provoking discussions and support throughout my doctoral study. I am grateful to Dr. Eric Hanson and Dr. Herman Lam for their suggestions and being on my supervisory committee. I would like to thank Dr. Nabil Kamel and Dr. Jose Principe for serving on my supervisory committee.

I thank Sharon Grant, the secretary of the Database Systems Research and Development Center, for her cheerful and efficient help throughout my stay at the Center. I would like to thank my colleagues and friends for their stimulating discussions and for making my stay very pleasant and enjoyable.

My special thanks go to my parents who have been patiently waiting for my graduation and to my wife Sireesha for being cooperative and sharing the burdens.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

GRAPH-BASED RULE AND TRANSACTION EXECUTION IN A PARALLEL ACTIVE OBJECT-ORIENTED KNOWLEDGE BASE MANAGEMENT SYSTEM

By

Ramamohanrao Sri Jawadi

December 1994

Chairman: Dr. Stanley Y. W. Su
Major Department: Computer and Information Sciences

The need for user-defined execution orders (or control structures) for rules is well recognized by researchers of active database systems. Priority-based approaches have been used to specify and enforce control structures among event-condition-action (ECA) rules. However, due to the fact that fixed priorities are assigned to ECA rules independent of different contexts in which they may be triggered, the existing approaches are not able to allow rules to be executed following different control structures when they are triggered by different events. In a priority-based approach, the control specification is mixed with the rule specification which makes the control difficult to understand and modify. More flexible and expressive control structures are needed for rules in advanced database applications such as CAD/CAM, CIM and Flexible Manufacturing Systems. The rule control mechanism must also be incorporated in the object-oriented data model so that control structures among rules can

be modeled uniformly. Since rules can be activated by operations in database transactions and their executions need to be incorporated in the transaction framework, a powerful transaction model is needed to deal with complex control structures among rules. Unfortunately, the variants of the nested transaction model used in several existing active systems are not expressive enough to handle rules with complex control structures in a uniform fashion. In this thesis, we introduce the concept of rule graphs in which control structures among rules are clearly separated from rules. We also separate the event part from the condition-action parts and associate it with a 'rule graph' which represents the control structure among a set of rules. Using rule graphs, the same set of rules may follow different control structures when they are triggered by different events. We uniformly extend an object-oriented knowledge model, which models rules as objects, to capture control relationships among rules uniformly. A graph-based transaction model is used to model graph-based control structures among database operations and rules in a uniform fashion. The proposed object-oriented knowledge model, rule and transaction modeling and execution techniques have been implemented and verified on a shared-nothing multiprocessor computer nCUBE2 which exploits the parallel execution properties of independent rules and DB operations with a graph-based control structure.

Key Words: Active object-oriented DBMS, rule, rule control, expressive transaction model, parallel and distributed implementation.

CHAPTER 1
INTRODUCTION

Event-condition-action (ECA) rules are used in active database management systems for a variety of purposes such as constraint enforcement, data derivation, triggers and alerters, version control and monitoring [Chak89, Hans93]. The event-part of an ECA rule specifies an event (or a set of events) which can trigger the rule, and the condition-part specifies a condition to be satisfied for executing the action-part. In practice, ECA rules are used for monitoring a variety of events that occur in a database environment and reacting with appropriate actions automatically. For example, a set of integrity constraints which check the consistency of a data item can be evaluated automatically whenever the data item is modified. In traditional database management systems (DBMSs), these constraints are explicitly coded in application programs. It is redundant to code such tasks in all the needed application programs (e.g., a constraint that checks the consistency of a data item has to be coded in all the application programs that modify the value of the data item). Also, it is difficult to modify the tasks that are buried in a number of application programs. For example, assume that the consistent range of values for a data item is modified. This modification will have to be incorporated in the corresponding constraint coded in many application programs. It is therefore advantageous to define such tasks as ECA rules. Rules get activated (or triggered) automatically, whenever that data item is modified, to check the constraints and take appropriate actions. The tasks represented by rules can be modified very easily also. For example, if the consistent range for a data item needs to be modified, it is enough to modify the corresponding

rule leaving application programs intact. Database systems coupled with rules are called *active database systems.* Although, we have illustrated the usage of ECA rules for only constraint enforcement, , they can be used in the same way for a variety of purposes mentioned earlier.

As described above, in active DBMSs, tasks that have to be performed automatically whenever some events occur, are defined as ECA rules. The event which causes a rule to be triggered is called *trigger operation* or *event.* A trigger operation can be a database operation (e.g., update to a data item), user-defined operation (e.g., hiring an employee), or an external signal (e.g., system clock). A trigger operation can be defined to trigger multiple rules, (e.g., when multiple constraints need to be checked after an update operation). A rule can be triggered by multiple trigger operations (e.g., when a constraint has to be checked in a number of situations). A triggered rule is executed at an appropriate time depending on its *trigger time* (or coupling mode) [Chak89]. Executing a rule means that evaluating the condition, and performing the action only if the condition is met.

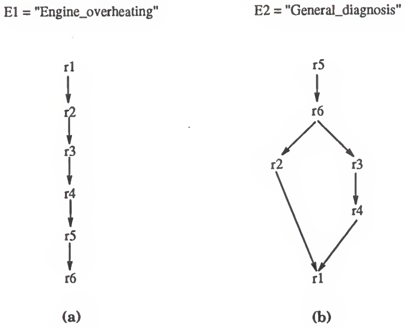
Often, rules triggered by a trigger operation are semantically inter-related and they need to be executed in a specific (partial) order. For example, in an Automated Car Manufacturing System, an event called "Engine overheating" can cause several tests, such as testing the hose, testing the radiator for leaks, testing the coolant level, etc., to be made automatically. According to the likelihood of the causes of the overheating problem, they are performed in a specific order, e.g., the hose must be checked first, radiator checked next, coolant level third, etc. In the majority of the active systems, such order is specified by assigning priorities to rules (e.g., by assigning a higher priority to hose-testing-rule than radiator-testing-rule) [Ston88, Hans89, Agra91, Chak94]. In this thesis, we use the phrase "rule control structure"

(or simply "rule control") to refer to the partial order (or a directed acyclic graph) to be followed by a set of rules triggered by a trigger operation.

In many advanced applications, a rule control is much more complex than what can be captured by priorities. Consider a set of rules which can be triggered by different events. These rules may have to follow different control structures when they are triggered by different events because different events may have different semantics. This is illustrated by the examples given below. In these examples, we shall separate the event part from C and A parts of the ECA rule, and associate it with a set of condition-action pairs (or CA rules) having some control structure among them. This means the event part not only decides which CA rules must be triggered but also the order of their execution.

Example 1: Consider the CA rules r1-r6 shown in Figure 1.1 which are to be triggered by different events in an Automated Car Manufacturing application. Two such events are "Engine overheating" and "General diagnosis". These two events occur in two different contexts, the former event occurs when the engine-temperature exceeds some limit during the test-phase of a car. It triggers a set of CA rules to fix the problem immediately. The latter event occurs just before a routine diagnosis of an engine begins and it triggers a set of CA rules to diagnose the engine failure. Since the two events have two different semantics, CA rules may follow two different control structures when they are triggered by these two events. We assume that the control structure shown in Figure 1.1.a is followed when it is triggered by the event "Engine overheating" and the control structure shown in Figure 1.1.b is followed when it is triggered by the event "General diagnosis".

Example 2: In an employee database, consider the following CA rules r1, r2, r3 and r4 which are triggered by two trigger operations namely "Promotion from Engineer to Designer" and "Promotion from Junior Manager to Manager".



r1: If hose broken Then Change hose

r2: If fan belt loose Then Adjust belt

r3. If radiator without coolant Then Add coolant

r4: If radiator leak Then Fix leak

r5: If engine valve noise Then Adjust valves

r6: If cylinder compression low Then Start diagnosis procedure for compression problem

Figure 1.1. Rules for Diagnosing an Engine

```
r1: If employee violated the company code for more than ten times
    Then cancel the promotion and abort.
r2: If employee has a Ph.D degree
    Then evaluate a new salary on Ph.D scale.
r3: If employee has more than 15 years of experience
    Then add a special bonus to the salary.
r4: If employee has exceeded the 100000 salary limit
```

```
    Then make the salary 100000.
```

Also, consider the following two CA rules r5 and r6 which promote employees in management based on their experience and degree automatically. They are triggered by an update to employee's experience or degree.

```
r5: If a Junior Manager has more than 10 years of experience and
       has a Ph.D. degree
    Then Transfer the Junior Manager to Administration office
         AND Change his/her designation to Junior Director


r6: If a Manager has more than 5 years of experience and
       has a Ph.D. degree
    Then Transfer the Manager to  Administration office
         AND Change his/her designation to Director
```

When an Engineer is promoted to Designer, rules r1-r4 need to maintain the following control relationships during their execution : (i) rule r1 has to precede all the other triggered rules (r2, r3, r4) because if the employee has violated the company code for more than 10 times, the promotion is canceled, (ii) rule r2 has to precede r3 because r2 evaluates a new salary and r3 adds a bonus amount to the salary, and (iii) r4 has to follow all the rules because it checks the salary for company's maximum salary limit. The resultant control structure for the above CA rules when they are triggered by "Promotion from Engineer to Designer" is shown in Figure 1.2.a.

In case of "Promotion from Junior Manager to Manager", all of the above control relationships apply except the second one (on rules r2 and r3). Note that, according to rule r5, if a Junior Manager has more than 10 years of experience and has a Ph.D. degree then he/she is transferred to Administration office and becomes a Junior

Director. Therefore, for a Junior Manager, only one of the conditions of rules r2 and r3 is true, i.e., a Junior Manager either has less than 10 years of experience or has a Ph.D degree. Hence, rules r2 and r3 need not follow any order because only one of them is going to change employee's salary. They can be executed in parallel. The resultant control structure for the rules when they are triggered by "Promotion from Junior Manager to Manager" is shown in Figure 1.2.b.

It can be observed that the CA rules r1-r4 have to follow two different control structures when they are triggered by two different operations namely "Promotion from Engineer to Designer" and "Promotion from Junior Manager to Manager".

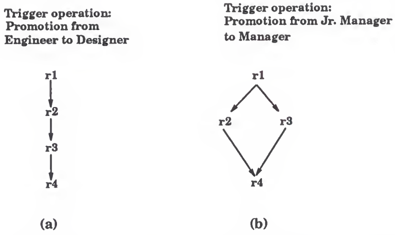


Figure 1.2. Control Structures among Employee DB Rules

Example 3: Another application domain where rules have different control structures in different contexts is "Automated Product Assembly and Disassembly System". During the assembly, several components have to be assembled following a specific partial order (or control structure) and during the disassembly the partial order often needs to be reversed. For example, a battery is charged to 15 volts first, tested for its performance and cleaned before it is assembled in a car. When it is disassembled from a car, the tasks are done in the reverse order, i.e, the battery is

cleaned first, it is tested for its performance and charged to appropriate voltage before it is stored in a battery store. The rules that clean, test and charge the battery are triggered by events "Assemble battery" and "Disassemble battery".

From the above examples, it is clear that the same set of CA rules may have to follow different control structures when they are triggered by different events. However, by assigning fixed priorities to ECA rules it is not possible to capture the above semantics because ECA rules follow the same control structure (specified by the priorities) even when they are triggered by different events. More *flexible* rule control is necessary to meet the requirements of the above applications.

Other disadvantages with the priority-based approach are: i) it is difficult to understand (or modify) the rule control because it is mixed with rules, ii) it is difficult to express the independent nature of a set of rules, and iii) it is difficult to add new rules with desired control requirements. For example, consider the following ECA rules with their numeric priorities in the parentheses: R1(1), R2(2), R3(3), R7(7), R8(8) and R10(13). R1, R2 and R3 are specified to be triggered by an event called E1, and R7, R8 and R10 are specified to be triggered by an event called E2. Consider the case where a new rule R20 whose event is "E1 or E2", has to be inserted into the rule base. Its control requirements are that it has to be executed before R2 when it is triggered by E1 and after R7 when it is triggered by E2. Note that there is no numeric priority (which is less than 2 and greater than 7) that can satisfy the above control requirements. Hence, there is a need for a more flexible and expressive rule control mechanism than the priority-based approach.

In another research domain, the database technology itself has been actively moving towards the object-oriented (OO) approach which is more suitable than the traditional attribute-based approach for modeling and processing the data found in

advanced application domains such as CAD/CAM, CASE, CIM, command, control, communication, office automation and multi media [Hamm81, Bato85, Bane87, Hull87, Su89]. In an OO model, all entity types of the real world such as 'Person', 'Student' are modeled as object classes and relationships among them are modeled as semantic associations. For example, the fact that every student is a person is modeled using a *generalization* association between 'Person' and 'Student' and the fact that an appointment order consists of an employee and a department in which he is appointed, is modeled by an *aggregation* association between 'Appointment-order' and 'Employment' and an aggregation association between 'Appointment-order' and 'Department'. 'Person' is defined as a generalization of 'Student' and 'Faculty' and 'Appointment-order' is defined as having aggregation associations with 'Employee' and 'Department'.

Several OODB research efforts have incorporated rules in their OO framework. Many of them adopted the uniform approach in which rules are also modeled as *objects* just like other objects in the database [Daya88, Su89, McCa89, Diaz91, Gatz92]. Others have modeled rules as *methods* [Huds89, Geha91, Beer91]. Diaz et al. [Diaz91] and Anwar et al. [Anwa93] describe the process of modeling rules as objects as more uniform and efficient than other approaches. However, so far rules have been modeled as independent objects, and, to our knowledge, the control structures among rules have not been incorporated in the existing OO data models.

In this thesis, we address the incorporation of a flexible and expressive rule control mechanism in an object-oriented knowledge base management system (OOKBMS). Some important requirements for the incorporation are as follows:

- *A flexible and expressive rule control mechanism* which enables rule designers to define complex control structures among CA rules and allows them to

follow different control structures when they are triggered by different trigger operations.

- An *object-oriented knowledge model* to uniformly model CA rules and the control structure among them. The existing OODB framework should be extended uniformly to capture the association between an event and a set of structured CA rules. An OO rule language (as part of an OO knowledge definition language) must be designed to enable a rule designer to specify events and their association with a set of structured CA rules.

- An *expressive transaction model* to incorporate the structured CA rules in a transaction framework. The control structure among rules can be quite complex (not just a linear structure or a tree structure but can be an acyclic graph). Since rules can be triggered by the operations in a DB transaction and rules in turn generate DB operations, their execution must be incorporated into the DB transaction framework. The traditional flat transaction model [Haer83] and the variants of the nested transaction model [Moss85] are not able to incorporate rules with the graph-based control structure in their transaction framework uniformly. A more expressive and powerful transaction execution model (e.g., a graph-based model) is needed.

An OOKBMS is very complex compared with an attribute-based DBMS because of its rich semantics and modeling abilities. Incorporation of rules in an OOKBMS adds to its complexity and its performance would be deteriorated significantly if care is not taken. One of the promising solutions for better performance of a OOKBMS is to implement it on a high-performance parallel computer. However, no earlier research work has addressed the issues in the parallel implementation of an active

OOKBMS. In this thesis, we also propose a software architecture for a parallel active OOKBMS server and describe its implementation on an nCUBE2 computer.

The remainder of the thesis is organized as follows. In Chapter 2, the related research work is presented. In Chapter 3, we introduce the concept of rule graph in which an event is associated with a rule graph - a set of CA rules having a graph-based control structure, and present a set of expressive control constructs to define rule graphs. In Chapter 4, we extend an object-oriented knowledge model called OSAM*, which models rules as objects, with a set of control associations to model control relationships among rules. This is analogous to the use of semantic associations to model semantic relationships among data objects. We also design an OO rule language which enables a rule designer to define rules and their control structures as a part of the knowledge base definition. During run time, the control structures among rules are automatically enforced by the graph-based execution model. In Chapter 5, we present a graph-based transaction model which models rules as subtransactions and their control structure as the graph structure among subtransactions. It also models the nested triggering of rules using the hierarchical control structure. In Chapter 6, we present a client/server-based architecture and an asynchronous execution model for the implementation of the proposed OOKBMS on an nCUBE2 computer. In Chapter 7, we describe the implementation of the parallel execution model on nCUBE2 and evaluate the system performance in terms of speedup and scaleup. Finally, in Chapter 8, we present the summary and the conclusion of this thesis, and the future work.

CHAPTER 2
SURVEY OF RELATED RESEARCH

The main goal of our research is to provide a flexible and expressive rule control mechanism for an active object-oriented knowledge base management system and implement it on a shared-nothing parallel computer. In this chapter, we shall survey several active and/or object-oriented database systems with respect to (i) rule control structure, (ii) object-oriented data model, (iii) transaction model, and (iv) parallel rule execution.

2.1 Rule Control Structure

Rules in active database environments often have complex control structures among them which impose a partial or total order on their execution when they are triggered. The majority of the existing active database systems adopt a priority-based approach in which a rule designer specifies such control structures among rules using rule priorities. We describe the priority-based approaches of HiPAC, PRSII (POSTGRES Rule System), Ariel and Starburst and explain why there is a need for a more flexible and expressive rule control mechanism.

High performance active database (HiPAC). HiPAC [Chak89] was a research endeavor on active and time constrained data management. Rules are treated as first class objects and each rule is an instance of the system-defined rule class. Rules in HiPAC are defined by specifying a rule identifier, an event, a condition, an action, timing constraints and rule attributes.

An event can be a data event, the beginning of a transaction, the end of a transaction, a temporal event or an abstract event. A condition, in addition to specifying a collection of queries to be evaluated when an event occurs, also specifies a coupling mode between the triggering transaction and condition evaluation. This coupling mode denotes *when* the condition is to be evaluated with respect to triggering transaction. The action part of a rule specifies an operation to be performed when the rule is triggered and the condition evaluates to be true. Furthermore, a coupling mode is also provided to determine when the action is to be executed with respect to the triggering transaction. The first version of HiPAC did not support any form of rule control. When multiple rules are triggered, it is assumed that all the rules are executed concurrently in a random fashion [Zert90]. However, extensions of HiPAC have used numeric priorities to control the execution of rules [Chak94]. In the implementation, when multiple rules are triggered, HiPAC spawns multiple threads each of which processes a rule. It assigns the rule priorities to the corresponding threads so that rules are processed according to their priorities.

PRSII (POSTGRES rule system). PRSII is a rule system which is quite naturally embedded in a general-purpose database manager (POSTGRES). A more detailed description is given in [Ston88]. POSTGRES supports a query language, POSTQUEL, which borrows heavily from its predecessor, QUEL. The main extensions are language constructs to deal with procedural data, extended data types, rules, inheritance, versions, and time. PRSII allows any POSTQUEL command to be tagged with three special modifiers which change its meaning. Such tagged commands become *rules* and can be used in a variety of applications. The three special modifiers are **always**, **refuse** and **one-time**. PRSII supports numeric priorities for

rule control. Priorities are unsigned integers in the range 0 to 15 and may optionally appear at the end of a command, e.g.,

```
always replace EMP(sal = 1000)
where EMP.name = "Mike"
at priority = 7
```

If a priority is not specified by a user, then PRSII assumes a default of 0. When more than one rule can produce a value, PRSII would use the rule with the highest priority. Hence the user can optionally specify any collection of tagged commands that he/she introduces, and the rule with the highest priority will be selected in case of a conflict. If multiple rules have the same priority, then PRSII chooses to implement random semantics for conflicting rules, and the result specified by any one of them can be returned. Another active DBMS called Ariel [Hans89] also adopts the same rule control mechanism (i.e., based on numeric priorities).

Starburst. Starburst has a facility for creating and executing *database production rules*; it is fully integrated into Starburst extensible relational system at the IBM Almaden Research Center [Wido91]. Production rules are persistent in Starburst like other active DBMSs and are created using a *rule definition language.* As users and applications interact with the data in the database, rules are triggered, evaluated, and executed automatically by the *database rule processor.* The design goals of the Starburst rule system were to design a rule definition language for defining a flexible execution semantics and the implementation and integration of the rule processor using the extensibility features of Starburst.

In Starburst, the syntax for creating a rule is

create rule name **on** table

when triggering operations

[**if**] condition

then action

[**precedes** rule-list] [**follows** rule-list]

The *triggering operations* are one or more of inserted, deleted, and updated (c1,...cn), where c1,...cn are columns of the rule's *table.* The optional *condition* is an arbitrary SQL predicate over the database. The *action* is an arbitrary sequence of database operations, including SQL data manipulation commands, data definition commands, and rollback. The optional **precedes** and **follows** clauses are used to partially order the set of rules: If a rule *r1* specifies a rule *r2* in its **precedes** list, or if *r2* specifies *r1* in its **follows** list, then *r1* has a higher priority than *r2.* Commands are also provided to alter, drop, activate, and deactivate rules. *Rule sets* may be created; each set contains zero or more values, and each rule belongs to zero or more sets.

The Starburst rule system provides mechanisms that can be used for database functions such as integrity constraints, derived data, situation monitoring and alerting, and as a platform for large knowledge-base and expert systems.

Motivation for an expressive and flexible rule control. In many advanced applications, the same set of rules have to follow different control structures when they are triggered by different events because different events have different semantics (see the examples given in Chapter 1). Priority-based approaches lack flexibility to allow rules to follow different control structures when they are triggered by different events. This is because in a priority-based approach, fixed priorities are assigned

to ECA rules and they have to follow the same control structure (resulting from the priorities) even when they are triggered by different events. Priority-based approaches also lack expressiveness to explicitly specify parallelism among rules, and to accommodate new rules with desired control requirements. The above facts form our motivation for an expressive and flexible rule control mechanism which not only captures control structures among rules but also allows them to follow different control structures when they are triggered by different events.

2.2 Object-oriented Modeling of Rule Control

Along another line of research, several OODB efforts have incorporated active rules in their OODB frameworks in a uniform fashion [Daya88, Huds89, Su89, McCa89, Diaz91, Geha91, Beer91, Gatz92, Anwa93]. Some of the above efforts incorporate rules as *objects* and others model them as *methods.*

Ode is an active OODBMS being developed at AT&T Bell laboratories [Geha91, Geha92]. In Ode, the active behaviour is supported by rules, in the form of constraints and triggers. Both constraints and triggers consist of a condition and an action. They are defined within a class definition similar to member functions (methods). Constraints are used to maintain the notion of object consistency and hence are applicable to all instances of the class in which they are declared. Triggers, on the other hand, are used for monitoring database conditions other than those representing consistency violations and are applicable only to those instances specified explicitly by the user at run time. Similar to Ode, Beeri and Milo also model rules as methods [Beer91]. They merge the concepts of active database, object-oriented database and nested transaction. Both rules and transactions are modeled as methods and all methods are potential triggering events. Hence, a transaction or a rule can trigger set of rules and triggered rules in turn can trigger rules, forming a method invocation

hierarchy. This hierarchical control structure is captured by the tree control structure of the nested transaction model (NTM).

Focusing on the uniform incorporation of rules in an OO framework, several other works incorporate them as first class objects in OODBMS environments. ADAM [Diaz91] is an active OODB implemented in PROLOG. It focuses on providing a uniform approach to the treatment of rules in an OO environment. Both events and rules are treated as first class objects which are created, deleted and modified in the same fashion as other objects. The rules are incorporated by using object-based mechanism, i.e. an object's definition is extended to indicate which rules to check when the object raises an event. Thus each class structure is augmented with a class-rules attribute. This attribute has the set of rules as its value which needs to be checked when the class raises an event. Sentinel [Anwa93] also incorporates rules as first class objects. Sentinel is an active OODBMS being developed at Database Systems Research and Development Center, University of Florida. In Sentinel, the object classes which need to be monitored are called reactive classes. The rule objects fall in notifiable objects category which is informed of the events generated by reactive objects. And rules take appropriate action depending upon the notified event. Sentinel models events also as objects hence the properties (state, structure, and behavior) of events can be modeled.

Although, the above systems have taken different approaches for uniformly incorporating active rules in an OODB framework, there has not been enough focus on the incorporation of control structure among rules in the object-oriented model. For supporting an expressive and flexible rule control, an active OODBMS, needs the following enhancements : i) uniform incorporation of rule control relationships in its active object-oriented data model, ii) provision of an object-oriented rule definition language for defining rules and their control structures in an OO fashion.

2.3 Transaction Model and Trigger Times

The majority of active systems support a variety of trigger times to enable triggered rules to execute at different points during the triggering transaction's life time (sometime as a different transaction altogether). They also support nested triggering of rules, i.e., an operation of triggered rule in turn triggering another set of rules. Triggerings can be arbitrarily nested. There have been several attempts to integrate rule systems with database systems to make the resulting systems active. Most successful efforts are by extending the traditional transaction execution semantics to accommodate rule executions [Hsu88, Ston88, Chak89, Buch91, Beer91, Rasc91, Care91, Wido91]. We shall examine some transaction models used in the existing active DBMSs below.

POSTGRES [Ston88] uses a flat transaction model. In PRSII, unlike other active relational database systems, rule processing is invoked immediately after a modification to a tuple that triggers and satisfies the condition of one or more rules. The triggered actions are incorporated in the transaction by a linear expansion [Zert90] and are executed immediately (semantically equivalent to HiPAC's `immediate` coupling mode). In PRSII, triggered actions can occur on demand also (by a lazy evaluation). Rule actions in PRSII are arbitrary database operations. Hence, when a rule's action is executed, it may modify multiple additional tuples, each of which may trigger additional rules. Consequently, rule processing in PRSII is inherently synchronous, almost like a procedure call mechanism.

In Starburst [Wido91], rules are processed at *rule processing points.* There is an automatic rule processing point at the end of each transaction, and there may be additional user specified processing points within transactions. The semantics of rule processing points is based on transitions (arbitrary database changes resulting from

executions of SQL operations). The state change created by the user transaction is the first relevant transaction, and some rules are triggered by this transition. As triggered rule actions are executed, additional transitions are created which may trigger additional rules. Within a transaction, the rule processing can be initiated by commands to process a rule or a set of rules. During the rule processing, triggered rules are determined using the transition log, and they are stored in the local main memory. Tasks generated by rules are treated as subtransactions of the top-level transaction. The Starburst query processor is called to evaluate rule conditions and execute rule actions. Since the query processor is called to execute the rule conditions and actions, concurrency control for these operations is handled automatically.

HiPAC [Hsu88] uses the nested transaction model [Moss85] which has an expressive control structure to model active database transactions naturally. Rules are treated as subtransactions of the triggering transaction and a nested triggering of rules is naturally modeled by the nested control structure of the NTM. NTM's locking rules automatically maintain the correctness during the concurrent execution of rules. HiPAC provides a variety of coupling modes using which rules can execute i) immediately (immediate), ii) during the commit time of the top-level transaction (deferred) or, iii) as a separate transaction (detached). The immediate rule-tasks (the tasks generated by rules which are executed immediately) are modeled as subtransactions of the triggering task, deferred actions are modeled as subtransactions of the top-level transaction, and detached actions are modeled as separate transactions. Beeri and Milo's model [Beer91], Starburst [Ceri92] and Raschid, Selis and Delis' work [Rasc94] are some of the other works that use variants of NTM to incorporate the execution of rules in DB environments.

However, the tree control structure of NTM is not sufficient for modeling more general (graph-based) control structures among rules in active DBMSs. For example, NTM does not capture the control relationships among rules, because rules are modeled as *independent* subtransactions. Also, the triggering order of the deferred rules is not modeled since rules are modeled as independent subtransactions of the top-level transaction [Hsu88]. There is a need for a transaction model with a more expressive control structure (e.g., graph-based model) than the flat and tree-based control structures to specify more complex control structures among rules.

Although, not targeted to capture the rule execution semantics, there have been several works in the area of transactions with expressive control structure [Buch91, Wach91, Shet91, Heil91, Chry91, Atti92]. DOM [Buch91], ACTA [Chry91], and the ConTract [Wach91], the Carnot project at MCC [Atti92] are some of the systems which emphasize the requirement of control structure among tasks (or DB operations) of a transaction. DOM supports precedent constraints among siblings of a nested transaction to capture the control structure among subtransactions. ACTA introduces a variety of dependencies among transactions and provides a specification language. In addition, it has a variety of events beyond `commit` and `abort` to impose the inter-dependencies. the Carnot project also provides several constructs to define dependencies among tasks and adopts a finite state automata based algorithm for enforcing the dependencies. ConTract provides a parallel programming language model to define the control structure of a transaction. However, all of the above systems do not address the relationship between a transaction model and the graph-based rule execution. Furthermore, since most of the above models are designed for long-lived transactions, they do not maintain the traditional ACID properties [Haer83].

2.4 Parallel Rule Execution

Rule processing in active DBMSs represents a significant burden on the system performance. One promising solution is to implement the active DBMS on a parallel computer. However, the parallel execution semantics must be clearly defined for executing rules in parallel.

In the field of artificial intelligence (AI), there has been significant amount of research for executing rules in parallel. Most of the work [Forg82, Gupt87, Mira87, Stol84] parallelized the *match* part (conditions of several rules are evaluated in parallel) of the OPS5 algorithm [Forg81]. The performance improvement is limited because actions of the matched rules are still executed sequentially [Gupt87]. For executing rules in parallel (both conditions and actions), two important problems need to be addressed: i) maintaining control structures among rules during their parallel execution and, ii) guaranteeing the serializable execution of rules.

The works of Schmolze et al. [Schm90, Schm91], Ishida [Ishi91], and Kuo et al. [Kuo91] focus only on the serializable execution of rules and do not provide any constructs to specify control structures among rules. Complementing Ishida's work, a parallel production system called PARULEL [Stol91], provides meta rules to specify the control structures among rules and executes all rules in parallel by simply following the meta rules. This approach ignores the rule serializability which is important in DB environments. It can be observed that the above rule systems either support control structures among rules and ignore their serializable execution or vice versa. Another drawback of the above systems is that they rely on the static analysis of the rules (or the meta rules) to maintain the rule serializability which is less efficient in DB environments because execution of several rules cannot be interleaved.

Parallel rule execution have been discussed by some DB researchers also [Rasc94, Ceri92]. Raschid, Sellis and Delis [Rasc94] execute the triggered rules in a concurrent fashion. They use a lock-based approach to maintain the serializability among the rules. Ceri and Widom [Ceri92] describe mechanisms that allow rule processing to occur separately at each site and guarantee its correctness. They also describe mechanisms that include locking schemes, communication protocols, and rule restrictions which can be used in any parallel and distributed environment. Based on a given parallel or distributed environment and desired level of transparency, the mechanisms may be combined or may be used independently.

Although, several systems support different important features, there is no system which i) provides a flexible and expressive rule control specification and ii) offers parallel rule execution strategies that maintain the rule control, and at the same, time exploit maximum parallelism.

CHAPTER 3
FLEXIBLE AND EXPRESSIVE RULE CONTROL

In database environments, ECA rules are used for monitoring trigger operations (or events) and reacting with appropriate actions automatically. Trigger operations can be database operations (e.g., Update to Employee.Salary), user-defined operations (e.g., Promote a Manager), or external signals (e.g., system clock signals). They are associated with rules. A trigger operation can be associated with multiple rules. Whenever that operation occurs, all the associated rules are triggered (or activated) for execution. Multiple trigger operations can be associated with a single rule which means that the rule is specified to be triggered by the occurrence of any of these trigger operations. Typically an ECA rule, in addition to a trigger operation, consists of a "condition" and an "action". A rule execution involves evaluating the condition and performing the action if the condition evaluates to be true.

3.1 Rule Graphs

We introduce "rule graphs" for specifying flexible and expressive control structures among rules. In our approach, we separate the E part from C and A parts of ECA rule and associate it with a "rule graph" which represents the control structure among a set of condition-action pairs (or CA rules). A rule graph is a directed acyclic graph (DAG) in which a node represents a CA rule and a directed edge from CA rule r1 to CA rule r2 specifies that r1 has to precede r2 during the execution. An example rule graph is shown in Figure 3.1.

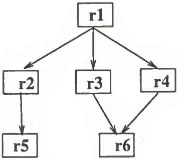


Figure 3.1. An Example Rule Graph

The semantics of the above graph is: execute r1 first and then execute r2, r3, r4 in any order (or in parallel if they are executed in a parallel computing environment), execute r5 after the execution of r2, and execute r6 after the execution of r3 and r4.

In our approach, trigger operation and trigger time (or coupling mode) are associated with a rule graph instead of a rule. Different trigger operations can trigger different rule graphs. Different rule graphs, may contain the same set of rules in different graph structures, or may have a common subset of rules in different subgraph-structures. This approach enables rule designers to specify control structures among rules in a more flexible fashion. For example, CA rules r1-r6 given in Figure 1.1 can be specified to follow two different control structures (shown in Figures 1.1.a and 1.1.b) when they are triggered by two different events, namely, "Engine overheating" and "General diagnosis", by simply defining two different rule graphs having the specified control structures and associating them with the corresponding events.

The rule graph representation is also expressive enough to specify complex control structure using its graph representation and to accommodate new rules with desired control requirements. For example, consider the following ECA rules (with the numeric priorities in the parentheses) R1(1), R2(2) and R(3) each of which has E1 as its event. They can be represented by a rule graph: $E1 : r1 \rightarrow r2 \rightarrow r3$ where r1, r2 and r3 represent condition-action pairs of R1, R2 and R3 respectively. And,

ECA rules R7(7), R8(8) and R10(13) each of which has E2 as its event, can be represented by the rule graph: $E2 : r7 \rightarrow r8 \rightarrow r10$. Consider the case where a new rule R20 with "E1 or E2" as its event needs to be inserted into the rule base and its control requirements are that it has to be executed before R2 when it is triggered by E1 and after R7 when it is triggered by E2. Note that, it is not possible to satisfy the above control requirements using numeric priorities. However, with rule graphs, the above control requirements can be satisfied by adding r20 which represents the condition-action pair of R20 to the above rule graphs as shown below:

E1: r1→r20→r2→r3;

E2: r7→r20→r8→r10

In this way more complex control semantics can be expressed using rule graphs. In addition, we show in the next section how the independent nature (or parallelism) of a set of rules can be expressed explicitly, using rule graphs.

In the graph-based approach, a CA rule can participate in multiple rule graphs as it can be triggered by multiple trigger operations. However, it is not efficient to specify a rule several times if it participates in several rule graphs. Our rule graph specification consist of only "rule names", and the actual rules (condition, action parts) are specified only once, independent of the number of rule graphs in which they participate. Actually, rule graphs and rules are defined in two different sections namely "RULE GRAPHS" and "RULES" of an object class definition which consists of several other sections (the OO model and the OO definition language are discussed in detail in Chapter 4). In this way, same rule definitions need not be repeated in a number of rule graphs. Furthermore, the rule control is clearly separated from rules which makes the complex control semantics of rules easy to understand and modify.

It can be observed that, with the rule graph representation, complex control structures among a large number of rules can be captured. To facilitate the specification

of rule graphs, we introduce three basic control constructs which are described in the following section.

3.2 Rule Control Constructs

The following control constructs are provided to define complex control structures among rules.

- *Precede construct:* This construct is useful to define an order among two CA rules. It is denoted by $\xrightarrow{s}$ and the semantics of r1 $\xrightarrow{s}$ r2 is that rule r2 should *sequentially* follow r1.
- *Precede-set construct:* This construct is useful to capture the situation in which a set of CA rules may execute (independently) in any order (may be in parallel in a parallel computing environment). It is denoted by $\xrightarrow{p}$ and the semantics of r $\xrightarrow{p}$ R, where R is a set of CA rules, is that the execution of r must precede the execution of all the rules in R. Rules in R can be executed independently (in a parallel processing system in *parallel*). Note that, this construct can be used to explicitly define the parallel execution property among rules.
- *Sync-at construct:* This construct is useful to capture a situation in which a single CA rule needs to wait for the execution of a set of CA rules. It is denoted by $\xrightarrow{y}$ and the semantics of R $\xrightarrow{y}$ r, where R is a set of CA rules, is that all the rules of R should *synchronize* at rule r.

Note that "$\xrightarrow{p}$" and "$\xrightarrow{y}$" constructs are equivalent to "$\xrightarrow{s}$" when R contains a single rule. However, "$\xrightarrow{s}$" is included as a control construct to allow the specification of a simple precedence relationship between a pair of rules.

3.3 Trigger Times

Besides *what* exactly CA rules in a rule graph perform, *when* exactly they perform is also very important. For example, rules which validate the access privileges of a user to perform a retrieval operation have to be executed *before* the operation, and rules which check the consistency of a new database state resulted from an update operation have to be executed *immediately after* the update operation, etc. This is facilitated by *trigger time* (or coupling mode) which is associated with each rule graph. A trigger time specifies when the associated rule graph should execute relative to the trigger operation (e.g., before, immediately-after, etc.). The trigger time is defined along with the trigger operation. Examples are:

`"Immediately_after Update to Engine.Temperature"`,

`"Before Engine_overheating()"`, etc.

The semantics of different trigger times supported by our rule definition language is as follows:

- *Before*: In case of "Before" as the trigger time, the triggered rule graph is executed just before the trigger operation. This is useful for performing several tasks defined by rules before an operation (e.g., checking the access privileges of a user before performing a retrieval operation).

- *Immediately-after*: In case of "Immediately-after" as the trigger time, the triggered rule graph is executed immediately after the trigger operation. This trigger time is useful for performing a set of inter-related tasks immediately after an operation (e.g, for processing several integrity constraints following a specific control structure, immediately after an update operation).

- *After*: In case of "After" as the trigger time, the triggered rule graph is executed just before the commit time. This is useful to delay a set of inter-related tasks defined by rules until the commit time, to make sure that they are performed only when the transaction will surely commit.

- *Parallel*: In case of "Parallel" as the trigger time, the triggered rule graph is executed as a separate transaction graph. This is useful for executing some rule graphs as separate transaction graphs in parallel to the triggering transaction graph.

The above trigger times provide the rule designer with the flexibility to specify one of the four different execution options for a rule graph.

3.4 Comparison with a Priority-based Approach

In this section, we shall compare our rule graph approach with the numeric priority approach using the Automated Car Manufacturing example described in the Introduction chapter. We shall use an abstract rule language which is similar to the rule languages used by HiPAC, PRS2 and Ariel. In this language, a rule specification consists of five components namely, `rule-name`, `event`, `condition`, `action` and `priority`. HiPAC, PRS2 and Ariel use different syntax for the specification of event, condition and action parts. We shall use methods (or functions) for specifying conditions and actions, e.g., a method used in the condition part returns 'true' or 'false' after performing a specific test on the database state, and a method used in the action part performs a specific database task or an external task if the condition returns true.

In the numeric priority approach, if the Automated Car Manufacturing application needs the rules r1-r6 shown in Figure 1.1 to follow two different control structures

shown in Figures 1.1.a and 1.1.b when they are triggered by two different events Engine_overheating and General_diagnosis respectively, the rule designer must define the rules and the corresponding priorities as given below:

```
Rule name :   r1
Event     :   "engine_overheating"
Condition :   hose_broken()
Action    :   change_hose()
Priority  :   1


Rule name :   r2
Event     :   "engine_overheating"
Condition :   fan_belt_loose()
Action    :   adjust_belt()
Priority  :   2

Rule name :   r3
Event     :   "engine_overheating"
Condition :   radiator_without_coolant()
Action    :   add_coolant()
Priority  :   3

Rule name :   r4
Event     :   "engine_overheating"
Condition :   radiator_leak()
Action    :   fix_leak()
Priority  :   4

Rule name :   r5
Event     :   "engine_overheating"
Condition :   engine_valve_noise()
Action    :   adjust_valves()
Priority  :   5

Rule name :   r6
Event     :   "engine_overheating"
Condition :   cylinder_compression_low()
Action    :   diagnose_compression_problem()
Priority  :   6
```

The above rules have different control semantics when they are triggered by the event "general_diagnosis". Therefore, they must be assigned different priorities when

they are specified to be triggered by "general_diagnosis". Note that, priorities assigned to r5 and r6 need not be changed even when they are triggered by "general_diagnosis", because r5 precedes r6 in both cases (viz. engine_overheating, general_diagnosis). Therefore, we just add "general_diagnosis" to the event parts of r5 and r6, leaving other parts of the rules unchanged, as given below.

```
Rule name :   r5
Event     :   "engine_overheating" OR "general_diagnosis"
Condition :   engine_valve_noise()
Action    :   adjust_valves()
Priority  :   5

Rule name :   r6
Event     :   "engine_overheating" OR "general_diagnosis"
Condition :   cylinder_compression_low()
Action    :   diagnose_compression_problem()
Priority  :   6
```

For the other rules, the corresponding new rules must be specified with the changes in the event and the priority as shown below.

```
Rule name :   r21
Event     :   "general_diagnosis"
Condition :   fan_belt_loose()
Action    :   adjust_belt()
Priority  :   7

Rule name :   r31
Event     :   "general_diagnosis"
Condition :   radiator_without_coolant()
Action    :   add_coolant()
Priority  :   7

Rule name :   r41
Event     :   "general_diagnosis"
Condition :   radiator_leak()
Action    :   fix_leak()
Priority  :   8

Rule name :   r11
Event     :   "general_diagnosis"
Condition :   hose_broken()
```

```
Action    :   change_hose()
Priority  :   9
```

In the rule graph approach, rules and their control structures are specified in two steps. In the first step, the conditions and actions of the rules are specified as CA rules. In the second step, the control structures among the CA rules are specified using rule graphs and each rule graph is associated with the appropriate event. The CA rules for the Car Manufacturing example are as follows.

```
Rule name :   r1
Condition :   hose_broken()
Action    :   change_hose()

Rule name :   r2
Condition :   fan_belt_loose()
Action    :   adjust_belt()

Rule name :   r3
Condition :   radiator_without_coolant()
Action    :   add_coolant()

Rule name :   r4
Condition :   radiator_leak()
Action    :   fix_leak()

Rule name :   r5
Condition :   engine_valve_noise()
Action    :   adjust_valves()

Rule name :   r6
Condition :   cylinder_compression_low()
Action    :   diagnose_compression_problem()
```

And, the control structures among the above rules is specified using the rule graphs RG1 and RG2 shown in Figure 3.2.

We shall compare the above two approaches namely numeric priorities and rule graphs with respect to (i) the specification of rule control, (ii) the insertion of a new rule, and (iii) the deletion of an existing rule.

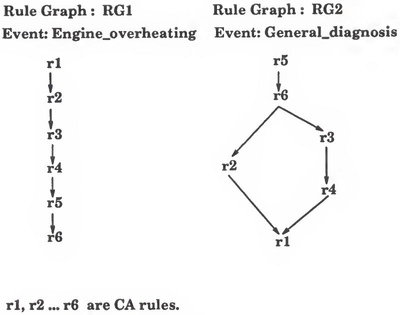


Figure 3.2. Control Specification using Rule Graphs

Specification of rule control. It can be observed that, in the priority-based approach condition and action parts of a rule need to be specified multiple times if the rule has different control semantics when it is triggered by different events. This redundancy is avoided in the rule graph approach, by defining the rule control structure at a higher-level as a rule graph using CA rules as building blocks, and associating the event with the rule graph.

Insertion of a new rule. We have illustrated earlier in the Introduction chapter that in the priority-based approach, sometimes it is not possible to insert a new rule with the desired control semantics. Whereas, using rule graphs a new rule with the desired control semantics can be inserted easily. The condition and action parts of the rule which is to be inserted can be specified in the rule base as a CA rule, and the control semantics of the rule can be incorporated in the rule graph specification.

Deletion of an existing rule. In the rule-graph approach, the deletion of a rule may cause a number of rule graphs to be restructured. However, the rule system makes sure that the rule designer restructures all the rule graphs in which the deleted rule is being referred, by giving a warning that there is an undefined rule being referred in some rule graphs. In the priority-based approach, it is easy to delete an existing rule because rule designer need not be concerned about repercussions on the rule control. However, sometimes the rule system may give unexpected results if the priority specifications of the existing rules are not readjusted after the deletion of a rule.

3.5 Effects on Data Model and Execution Model

Our objective is to provide a flexible and expressive rule control mechanism to an object-oriented KBMS. Therefore, the above explained rule control mechanism has to be uniformly incorporated in the OO data/knowledge model and the execution model of the OOKBMS.

In an object-oriented data/knowledge model, all the entities in a real-world application are modeled as objects and all the semantic relationships among the entities are modeled as semantic associations. Many OODB research efforts have uniformly extended the traditional OO data model with rules by modeling rules as objects. However, the control relationships among rules have not been modeled so far. We extend an OO knowledge model called OSAM* [Su89] with rules and their control relationships. The extended model is called OSAM*/P [Su94]. OSAM*/P models rules as objects and their relationships as control associations. In addition, we also design an object-oriented rule definition language to define rules and rule graphs.

In databases, all operations are performed in a transaction framework for consistency, correctness and reliability. Since rules are triggered by DB operations belonging to transactions and these rules in turn generate DB operations, rules also must be executed in a transaction framework. However, traditional transaction models based on linear (flat model) and tree control structures (nested model) are not expressive enough to model rules with graph-based control structures in a uniform fashion. We have designed an expressive graph-based transaction model which can model rules and their graph-based control structures naturally and uniformly.

The OSAM*/P knowledge model and the graph-based transaction model are discussed in detail in the next two chapters respectively.

CHAPTER 4
INCORPORATING RULE CONTROL IN AN OO KNOWLEDGE MODEL

In this chapter, we extend an OO knowledge model called OSAM* [Su89] with rule graphs. Also, we extend an OO rule language which is used for defining rule graphs, rules, trigger times and trigger operations in object class definitions.

4.1 OSAM* Knowledge Model

OSAM* is an object-oriented knowledge model, in which all the things of interest in an application world such as physical entities, abstract things, functions, events and processes are uniformly modeled as objects. In OSAM*, an application world is modeled as a *schema graph* which is basically a network of object classes and their associations. An example schema graph for a university database is shown in Figure 4.1. Rectangular boxes and circles represent two different categories of object classes, namely, entity classes (E-Class) and domain classes (D-class), respectively. The sole function of a D-class is to define a domain of possible values from which descriptive attributes of objects draw their values (e.g., integer, real, string). An E-class, on the other hand, forms a domain of objects which occur in an application's world (e.g., Person, Student). OSAM* also includes variety of semantic associations to model relationships among entity classes. Among them the important ones are *aggregation association* (A) and *generalization association* (G). In the above figure, the labels A and G stand for aggregation association and generalization association respectively. An A-association defines either a value attribute or a reference attribute depending upon whether its constituent-class is a D-class or an E-class respectively.

For example, an object class 'Transcript' is defined by three binary aggregation associations with object classes 'Student', 'Course' and 'Grade' (which are called constituent classes) respectively. Two A-associations define two reference attributes for 'Student' and 'Course' which are E-classes, and another A-association defines a value attribute for 'Grade' which is a D-class. A G-association specifies superclass-subclass relationship between two classes. For example, an E-class 'Person' is a generalization of the E-class 'Student' and is also a generalization of the E-class 'Teacher'.

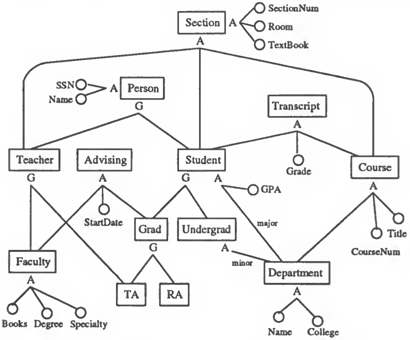


Figure 4.1. Example Schema of a University Database

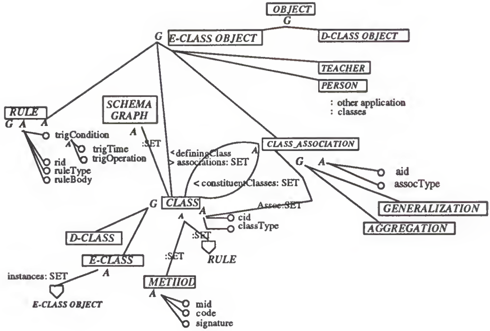
The structure and the semantics of different modeling constructs of OSAM* can be better understood by studying the OSAM* meta model [Lam94] which is a model of the OSAM* knowledge representation model. The OSAM* meta model models all the modeling constructs of OSAM* at a higher level, e.g., all associations such as aggregation and generalization are modeled as sub-classes of an object class called

CLASS-ASSOCIATION and all the structural and behavioral properties that are common to all class associations are captured by this class. The meta model views every thing of interest as an object. Figure 4.2 shows the OSAM* meta model. The objects are divided into two broad categories: E-CLASS OBJECT (for entities) and D-CLASS OBJECT (for attribute values). The former category models objects of interest to a data model (e.g., class, method, rule) and to an application (e.g., Teacher, Person). The identifiers for these objects are assigned by the system. The latter category models self-naming objects (e.g., integer, real) which serve as descriptive data of entity objects and/or other complex domain objects. In the meta model, OSAM*'s modeling constructs such as *classes*, *class_associations*, *methods*, and *rules* are defined as first-class objects. For example, the meta class named CLASS contains a set of objects (or instances) each of which is the definition of a class. From the meta model, one can see that the structure of a class consists of a class-id, a class-type, a set of class associations, a set of rules, and a set of methods (see Figure 4.2).

OSAM* has a knowledge definition language to define an application schema. In OSAM*, the class definition is the main building block of an application schema definition. An application schema is defined by a set of class definitions. An example OSAM* class definition is given below.

```
/* Each class definition consists of four sections: ASSOCIATIONS,
METHODS, RULES, and IMPLEMENTATIONS */

Entity_class  Student  is
  ASSOCIATIONS:
    specialization of Person;     /* Student is a subclass of Person */
    generalization of R.A., T.A.; /* Student is a superclass of R.A. and T.A. */
```



: SET represents set-valued attributes

is a connector (or pointer) to another class in the schema

< label of the link from defining class to constituent class. This is the default direction if no label is shown.

> label of the link from constituent class to defining class

Figure 4.2. The OSAM* Meta Model

```
  aggregation of
    public:                    /* Definition of public attributes */
      S_Name:  array[40] of char;
      enroll:  set of Course;  /* A student can enroll in a set of courses */
      Status:  array[20] of char;
    protected:                 /* Definition of protected attributes */
      SS#:     integer;
      Eligibility:  array[10] of char;

METHODS:                       /* The signature of methods */
    public:
```

```
      method eval_GPA() : GPA_Value;
      method inform_all_instructor() : void;

  RULES:
    Rule id: R1
      triggered after update major
      condition (major.name = "CIS" AND eval_GPA() > 2.0)
      action  ----
      otherwise suspend()

    Rule id: R2
      triggered after suspend()
      condition ( Eligibility = "No")
      action    inform_all_instructor();

  IMPLEMENTATIONS:                /* Actual coding of methods */
      method eval_GPA() : GPA_Value is
       {
        begin .....  end;
       }

      method inform_all_instructor() : void is
       {
        begin .....  end;
       }

END  Student;
```

It can be observed that in OSAM*, rules are modeled as first class objects (see the metamodel in Figure 4.2) and they are defined in the "RULES" section of a class definition. The event and the trigger time are defined along with condition and action parts of the rule, similar to the ECA rule structure. However, the existing OSAM* knowledge representation model does not support control structures among rules. We have extended the OSAM* knowledge model with a set of new rule associations to model rule graphs.

4.2 Model Extension

The extended model is called OSAM*/P. In OSAM*/P, rules are modeled as objects to be uniform with data objects. To support rule graphs, OSAM*/P must be able to model different control relationships among rules. We model control relationships among rule objects just like the way semantic relationships among data objects are modeled. Class associations such as generalization and aggregation are used to model semantic relationships among entity classes. Analogous to these association types, we introduce a set of *rule associations* (or control associations) to model different control relationships among rules. A rule graph is defined by a set of rules with different rule associations among them.

The structure and semantics of rule graphs can be more clearly defined using the meta model. As shown in Figure 4.3, we introduce a new subclass of the E_Class_Object called `RULE GRAPH` which is defined as having aggregation associations with a trigger condition (trigger time and trigger operation) and a set of `RULE_ASSOCIATION`s. In Figure 4.3, the extensions are shown using hyphenated boxes.) `RULE_ASSOCIATION` which is also modeled as a subclass of the E_Class_Object is defined as having aggregation associations with a defining rule and a set of constituent rules. `RULE ASSOCIATION` class has three different subclasses namely S, P

and Y. Each subclass models a specific control association between the defining rule and set of constituent rules. The semantics of the S, P, Y associations are as follows.

S-Association: This captures the semantics of the *sequential* construct explained earlier. For example, the S-association in Figure 4.4.a specifies that the execution of rule r9 precedes the execution of r10.

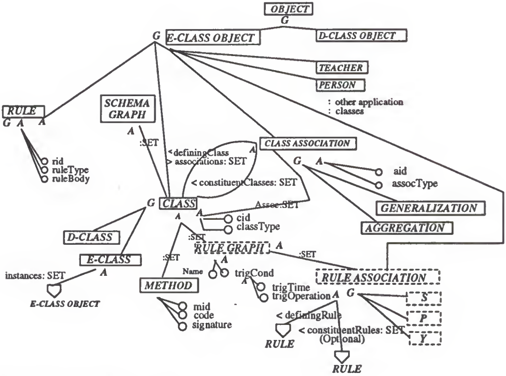
P-Association: This captures the semantics of the *precede-set* construct explained earlier. For example, the P-association in Figure 4.4.b specifies that the execution of rule r1 has to precede the execution of rules r2..r4 and they can be executed in any order (may be in parallel in a parallel computing environment).

Y-Association: This captures the semantics of the *sync-at* construct explained earlier. For example, the Y-association in Figure 4.4.c specifies that rules r5..r7 must be executed before r8 begins its execution.

In addition, we extend the class structure with rule graphs, so that rule control structures can be specified in an application schema. The `CLASS` in the meta model now consists of the specification of rule graphs in addition to other things. The semantics of all the other constructs (e.g., generalization, aggregation associations) remain the same.

The above extensions are made available to application designers by extending the OSAM* knowledge definition language. In OSAM*/P, an application schema is defined by a set of class definitions. However, the OSAM*/P class definition contains a new section called RULE_GRAPHS, in addition to ASSOCIATIONS, METHODS, RULES and IMPLEMENTATIONS. All the rule graphs that are triggered by operations on a class are specified in the RULE_GRAPHS section of that class. In rule graph specifications, only rule names are used to refer to the rules, and the rules

are specified in the RULES section of the class. OSAM*/P provides a rule language using which rule graphs and rules can be specified as a part of a OSAM*/P schema definition.



: SET represents set-valued attributes

is a connector (or pointer) to another class in the schema

< label of the link from defining class to constituent class. This is the default direction if no label is shown.

> label of the link from constituent class to defining class

Figure 4.3. The Meta Model Extended with Rule Graphs

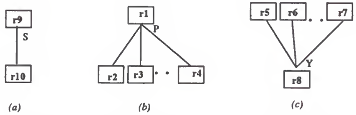


Figure 4.4. Control Associations among Rules

4.3 OSAM*/P Rule Language

4.3.1 Rule Graph Definition

The syntax and semantics of the rule associations which are used to define rule graphs are as follows:

Rule Association S: An "S" association is defined between two rules to specify a sequential execution order between them. Syntax for S is as follows:

```
rule r1:
    S: r2; /* r1 precedes r2 */
```

Rule Association P: A "P" association is defined between a rule and a set of rules to specify that the rule should precede the rules in the set during the execution. Syntax for P is as follows:

```
rule r1:
    P: r2, r3, r4; /* r1 precedes r2, r3 and r4 */
```

Rule Association Y: A "Y" association is defined between a set of rules and a rule to specify that the rules in that set should precede the rule during their execution. Syntax for Y is as follows:

```
rule r4:
    Y: r1, r2, r3; /* r1 r2, r3 precede r4 */
```

The general syntax for a rule graph is as follows:

```
rule_graph     <rule graph name>  is
triggered      <trigger_conds>
<rule name>:
       S: <rule name>;
       P: <rule name>, <rule name> .... ;
       Y: <rule name>, <rule name> .... ;
<rule name>:
       S: <rule name>;
       P: <rule name>, <rule name> .... ;
       Y: <rule name>, <rule name> .... ;
 :
 :
end            <rule graph name>;
```

The "rule_graph is" and "triggered" clauses must exist. The rule_graph body consists of a list of rules and their associations. Each rule can have S, P and/or Y associations with other rules. All associations are optional. The keyword "triggered" specifies the trigger conditions. Trigger conditions are defined as follows:

```
trigger_conditions: trigger_condition |
                    trigger_conditions, trigger_condition
trigger_condition:  trigger_time trigger_operation
```

The trigger_time can be one of the following: Before, Immediately-after, After and Parallel. Trigger times have the same semantics as explained in Chapter 2. A trigger operation can be a system-defined DB operation or a user-defined method. For example, the rule graph shown in Figure 3.1 can be defined as follows:

```
rule_graph  RG1  is
triggered   Before Update Employee.BasicSalary
     r1:
       P: r2, r3, r4;
     r2:
       S: r5;
     r6:
       Y: r3, r4;
end  RG1;
```

A rule graph specification contains only rule names and the actual rules are specified in the RULES section of the class definition. The syntax and semantics of a rule specification are given in the next section.

4.3.2 Rule Definition

The general syntax for a rule is as follows:

```
rule            <rule name>  is
condition       <rule_condition>
action          <statements>
otherwise       <statements>
end             <rule name>;
```

Every rule must have a rule name. The condition clause is optional. The action and otherwise clauses are also optional, but one of them must exist. The condition clause specifies a state of the database using a condition expression which evaluates to TRUE or FALSE. If the condition evaluates to TRUE, then the operation(s) in the

action clause is executed. If the condition evaluates to FALSE, the operation(s) in the otherwise clause is executed. We will now use a number of examples to describe and illustrate various constructs of a rule.

```
Example 1.  (This rule is defined in the class Person)
        rule          ex100  is
        condition     ssn < 0 OR ssn > 999999999
        action        Delete this;
        end           ex100;
```

The rule in Example 1 is based on the university database shown in Figure 4.1. This rule illustrates a simple constraint within a single class. It deletes the person with an erroneous social security number, when it is triggered.

The condition clause can be more complex e.g., it can be an association pattern expression. The association pattern expression is formed using the association operators "*" and "!". An "*" operator between two classes is used to identify these objects of these two classes that are associated with each other (i.e., objects of one class that are associated with some objects of another class). For example (see Figure 4.1), STUDENT * SECTION represents the STUDENT objects which are associated with some SECTION objects and vice versa (i.e., students who are enrolled in some sections and sections which have students). The non-association operator "!" identifies the objects of one class that are not associated with any object of another class and vice versa. The following rule illustrates the use of an association pattern expression in the condition clause. This rule enforces a constraint over multiple classes.

```
Example 2.  (Defined in the class Grad)
```

```
rule        ex101 is
condition   exist this in  this * Advising * Faculty[Degree = 'Ph.D.']
otherwise   DisplayMessage ("Grad's advisor doesn't have a Ph.D. degree")
            .AND. Delete this;
end         ex101;
```

This rule is named in the rule graph which is triggered after a Grad object is inserted. It specifies that "this" inserted Grad object must be associated with a faculty advisor who has a Ph.D. degree. Otherwise, the two operations in the otherwise clause are performed.

The action part of a rule can also be a complex expression. For example, a "context statement" can be used in the action part. A "context statement" is used to iterate through the objects in a class in order to process them. For example:

```
context  this * c:Advising  do
    Delete c
end context;
```

This context statement iterates through the objects in the class Advising. For each Advising object (bound by the range variable c) which is associated (* operator) with "this" object, it is deleted. The following rule illustrates the use of a context statement.

```
Example 3.  (Defined in the class Faculty)
            rule        ex202 is
            action      context  this * c:Advising  do
                        Delete c
```

```
        end context;
end     ex202;
```

This rule specifies that before deleting a Faculty object (specified in the trigger clause not shown here), all the Advising objects which are associated with this Faculty object need to be deleted first. In addition, more complex operators namely branch operators (AND, OR) are supported by our rule language to define more complex expressions that involve branching structures of multiple classes. For detailed description of these operators, the reader is referred to Alashqur et al. [Alas89]. The BNF rules for the OSAM*/P rule language is given in the Appendix.

CHAPTER 5
GRAPH-BASED TRANSACTION MODEL

In DB environments, all operations are executed in a transaction framework. Since DB operations of a transaction may trigger rule graphs and rules in a rule graph may in turn generate DB operations, the execution of rule graphs must be incorporated in a transaction framework.

Although, the execution of rule graphs can be implemented using the existing transaction models such as flat and nested transaction models, their linear and tree control structures are not expressive enough to model the graph-based control structures of rule graphs in a uniform fashion. For example, in a flat model, rules cannot be modeled as subtransactions executing in different control spheres, because a flat transaction is executed as a single block of operations in one control sphere [Zert90]. On the other hand, a nested transaction model provides a hierarchical structure of control spheres. The control sphere corresponding to a top-level transaction may contain a set of (sub)control spheres each corresponding to a triggered rule, and a control sphere corresponding to a rule in turn can activate different new control spheres within itself when an operation of that rule triggers some more rules. Control spheres can be arbitrarily nested depending on the nested triggering of rules. However, nested model executes rules in different independent control spheres and the control structures among rules cannot be modeled in a uniform fashion.

In this chapter, we first introduce a graph-based transaction model and then explain its ability to model rule graphs and different trigger times uniformly and finally explain a concurrency control scheme for the proposed transaction model.

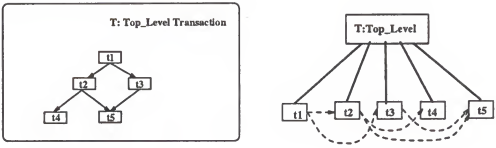
5.1 Graph-based Transactions

To model the control structure among the rules of a rule graph uniformly, it is logical to view a transaction as a control graph of subtransactions (each having its own control sphere) as shown in Figure 5.1.a. In this figure, boxes represent the spheres of control and the directed edges represent the control flow. This can be viewed as an extended tree control structure in which siblings can form a directed acyclic graph (DAG) depicting the (partial) order in which they should be executed. The extended tree structure is shown in Figure 5.1.b, in which solid lines represent parent-child relationships and hyphenated lines represent the control structure among subtransactions. Each subtransaction in turn can have a graph structure, which means that the tree can expand dynamically as the nesting depth of the triggered rule graphs increases. We shall call a transaction with the graph-structure a *transaction graph* (TG).

Using transaction graphs, the control structure of rule graphs can be incorporated uniformly and also different trigger times of a rule graph can be easily represented.

5.1.1 Modeling Control Structures of Rule Graphs

The structure of transaction graph is expressive enough to *uniformly* model graph-based control structures of rule graphs. For example, if a transaction T triggers the rule graph shown in Figure 5.2, rules in the rule graph are modeled as subtransactions of T, and the control structure of the rule graph is modeled as the control relationships among subtransactions. The resultant transaction graph are shown in Figure 5.3.



(a) (b)

Figure 5.1. Transaction Graph Model

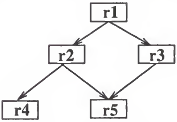


Figure 5.2. A Rule Graph

The graph model also captures the nested triggering of rule graphs. Rules of a triggered rule graph are modeled as subtransactions of the triggering rule and they have control structure among them. For example, if an operation activated by rule r2 (of the TG shown in Figure 5.3) triggers the rule graph shown in Figure 5.4.a, the rule graph is incorporated into the transaction graph structure as shown in Figure 5.4.b. The correctness criteria and the concurrency control method for the execution of a transaction graph is discussed in Section 5.2.

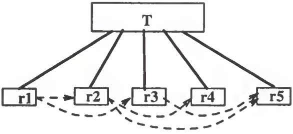
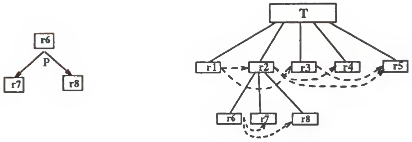


Figure 5.3. Active Rules as Part of a Transaction Graph



(a) (b)

Figure 5.4. Modeling Nested Triggerings of Rules

5.1.2 Modeling Trigger Times

Another important feature of the TG is its ability to support the control semantics of different trigger times. During the execution time, triggered rule graphs are included in the dynamically expanding control structure of a TG according to their trigger times. To include the rules with trigger time *after* at the end of the TG, the TG should have "begin" and "end" points. We add a *begin-task* which performs the initialization in the beginning of the control flow and a *commit-task* which commits the transaction atomically, at the end of the control flow as shown in Figure 5.5.a [1]. This can be represented using the tree form as shown in Figure 5.5.b. A triggered rule graph is added to the triggering transaction graph according to its trigger time, as explained below.

- In case of *before* or *immediately-after* as the trigger time, rules of the triggered rule graph are treated as subtransactions of the triggering task (or rule). For example, if an operation of t2 of the TG shown in Figure 5.5.b triggers the rule graph in Figure 5.6.c, it is coupled to the TG as shown in Figure 5.6.a.

[1] In the NTM, begin and commit tasks are embedded in the top-level transaction itself, they cannot be brought out because NTM cannot express that begin task has to be executed before all the other tasks and the end task has to be executed after all the other tasks. This is because NTM does not have any control structure among subtransactions.

Modeling rules as subtransactions of the triggering task enables the arbitrary nesting of rules, and the concurrency is automatically enforced by a locking scheme which will be explained in Section 5.2.4. However, to maintain the control structure among rules, rules have to be scheduled accordingly. The scheduler starts scheduling the rules before or immediately-after performing the trigger operation depending on the trigger time of the rule graph. Nevertheless, rules are executed as subtransactions of the triggering task in both cases.

- In case of *after* as the trigger time, the triggered rule graph is added just before the commit task. In the above example, if the trigger time is *after*, the rule graph shown in Figure 5.6.c is added to the triggering transaction, just before the commit task as shown in Figure 5.6.b and the triggered rules are treated as subtransactions of the top-level transaction. Note that it is different from treating the triggered rules as subtransactions of the triggering task. In addition, the *triggering order* of the *after* rule graphs is also captured. Assume that the task t2 triggers another rule graph with the trigger time "after", the rule graph is added just before the commit task but after the rule graph which was triggered earlier. In this way, the "after" rule graphs will be executed in the order they were triggered. The importance of maintaining such order is mentioned in [Hsu88].

- In case of *parallel* as the trigger time, the triggered rule graph is executed as a separate transaction in parallel with the triggering transaction. It is completely detached from the triggering transaction/rule in all respects except a causal relationship. A parallel transaction can be causally dependent or causally

independent of the parent transaction. In the first case, the failure of the parallel transaction causes the triggering transaction to abort and, in the second case, the failure does not affect the triggering transaction. However, the failure of a triggering transaction causes all triggered transactions including parallel transactions to be aborted.

It can be observed that, for all trigger times, the rule graph structure uniformly fits into the proposed transaction graph structure, which obviates the transaction manager treating rules differently from other subtransactions with respect to the scheduling, the correctness criterion and concurrency control techniques. It appears as though one transaction graph is expanding uniformly as the number of triggered rule graphs are being added at different levels of nesting.

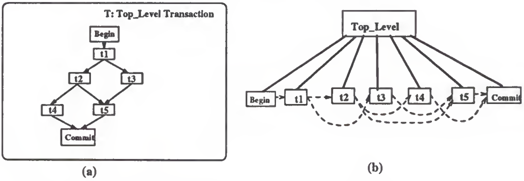


Figure 5.5. Adding Begin and End Points to a Transaction Graph

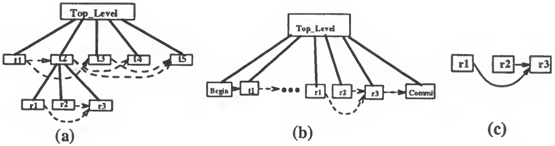


Figure 5.6. Coupling a Rule Graph to Transaction Graph

5.2 Concurrency Control

In a TG model, rules maintain atomicity and isolation, in addition, they need to maintain the partial order defined by the control structure. A random execution order can lead to semantically incorrect results. In this section, we define a correctness criterion for the concurrent execution of rules within a TG and describe a scheduling algorithm to enforce the control structure.

5.2.1 Correctness Criterion

In our model, each task (transaction or rule) is executed in its own control sphere. Transactions are executed in different transaction-level control spheres. Each such transaction-level control sphere may generate another set of rule-level control spheres when a set of rules are triggered and being executed within the transaction control sphere. Each rule-level control sphere may in turn generate a set of rule-level control spheres when an operation of the rule in turn triggers another set of rules.

At the transaction-level, the correctness criterion for parallel/concurrent execution of several TGs is the standard serializability [Gray93] which states that the interleaved (concurrent) execution of several TGs is correct when it is equivalent to some serial execution. It means that all transaction-level control spheres are clearly isolated from one another. Within a transaction, the control structure among subtransactions (or rules) plays a role in defining the correctness. The serializable execution of rules alone is not sufficient because an arbitrary serial order can violate the control structure among rules. For example, for the rule graph shown in Figure 5.2, the order $r1 \rightarrow r4 \rightarrow r2 \rightarrow r3 \rightarrow r5$ is not correct. The concurrent execution of rules is *correct* if and only if the resulting order does not violate the control structure of the rule graph. The correct execution orders for the rule graph shown in Figure 5.2 are

$r1 \rightarrow r2 \rightarrow r3 \rightarrow r4 \rightarrow r5$ and $r1 \rightarrow r3 \rightarrow r2 \rightarrow r4 \rightarrow r5$. To be more general, the concurrent execution of rules is correct when it is equivalent to some *topological order* [Corm90] of rules. Since the above correctness criterion requires a topological serial order, we shall call it *topological serializability.* To summarize, the requirements of the topological serializability are: (i) the execution of a rule must be atomic, (ii) the execution of a rule must be isolated from the execution of other rules in the system, and (iii) the execution of rules in a rule graph must be equivalent to a sequential execution of the rules in the topological order specified by the rule graph. The same correctness criterion applies to both rules in a rule graph and tasks in a transaction graph. In the remainder of this chapter, we explain scheduling algorithms and locking rules for processing rules in a rule graph. However, the same techniques apply to tasks in a transaction graph also because they have the same control structure as rules in a rule graph.

To maintain the topological serializability during the execution of a rule graph, it is sufficient to execute rules one after another in a topological order, nevertheless at the expense of potential concurrency among independent rules. For example, rules r2 and r3 in the rule graph shown in Figure 5.2 can be executed in any order without violating the topological order. In this example, rules r2 and r3 are said to be independent of each other. A set of rules can be executed in a concurrent fashion only if they are *independent* of each other. In general, a rule in a rule graph is independent if it does not have any incoming edge or its *indegree* (number of incoming edges) is equal to zero. For example, in the rule graph shown in Figure 5.2, r1 is independent and the rules r2, r3 become independent only after the completion of r1 and they can be executed concurrently.

5.2.2 Topological Scheduling

To maintain a topological order, rules in a rule graph can be executed in a topological order sequentially, nevertheless, at the expense of concurrency among independent rules. To achieve concurrency, rules in a rule graph can be divided into *topological groups* such that the rules in each group are independent of each other and can be executed concurrently. The topological groups can be formed as follows. All the rules in a rule graph can be given a *level* number such that every rule's level is lower than the levels of all its *successors.* Rule r2 is a *successor* to rule r1 if there is a directed path from r1 to r2 in the rule graph, and r1 becomes r2's *predecessor*[2]. The rules can be partitioned into topological groups by clustering them according to their level number. For example, the groups for the rule graph in Figure 5.2 are (r1:1),(r2:2, r3:2),(r4:3, r5:3). The number beside a rule indicates its level in the rule graph. The algorithm which gives the topological groups for a given control graph is given below.

```
Algorithm Topological-grouping
Input: A control graph
Output: Groups with levels

BEGIN
   Initialize v.indegree for all vertices (e.g. by a depth first search);
   level = 1;
   FOR i= 1 TO n DO    /* n is the total number of tasks */
      IF v(i).indegree = 0
```

[2]Note that, this is different from child-parent or ancestor-descendant relationships, which are denoted by solid lines in the tree representation used.

```
    THEN  v.level = level;
          put v(i) in Queue;

  REPEAT
    remove vertex v from the front of the Queue;

    FOR all edges (v,w) DO
      w.indegree = w.indegree - 1;
      IF w.indegree = 0
      THEN w.level = v.level + 1;
           put w in the rear of the Queue;

  UNTIL Queue is empty
END;
```

Within each topological group, the rules can be executed in a random order, however the serial order of the topological groups is maintained. Although the sequentiality among the groups may affect the concurrency among rules, it is necessary for maintaining the correctness. The concurrency can be further ameliorated by executing the rules *asynchronously*, which we shall discuss in the next section. The following algorithm schedules the rules of a rule graph in a topological order and exploits the parallelism among independent rules.

Repeat

Select: This step selects the set of independent rules which is unique for a given rule graph. As shown in the topological_grouping algorithm, the independent rules are identified by the indegree.

Schedule: This step assigns the independent rules to appropriate processing nodes for concurrent execution[3]. The isolation among rules is maintained by a locking scheme which will be explained later in Section 5.2.4.

Wait: This step synchronizes all the rules scheduled in one iteration by waiting for their completion. In the actual implementation, which will be discussed later in Chapter 7, the transaction manager switches to another transaction during the `wait` step, thereby interleaving the execution of multiple transactions.

Remove: This step removes all the completed rules as well as the outgoing edges of these rules from the rule graph, so that the `schedule` step gets the next set of independent rules in the succeeding iteration.

Until the rule graph is completely executed

Theorem 1: *The above algorithm schedules the rules of a given rule graph without violating the control structure.*

Proof: *We prove the theorem by contradiction. Assume that there is a scheduled rule r that violates the control structure, which implies that it is scheduled before the completion of r's predecessor say r1. According to the predecessor definition there is a directed path from r1 to r in the rule graph. This implies that there is at least one incoming edge to r when it is scheduled which means r's indegree is not zero. This contradicts with the hypothesis that the algorithm schedules only independent*

[3]On a uniprocessor machine, rules can be executed as child processes or threads.

rules. Hence the proof. Note that the wait-step waits until the completion of all the scheduled rules and then the remove-step removes the rules and the outgoing edges from the rule graph.

5.2.3 Asynchronous Scheduling

In the above scheduling algorithm, the `wait` step waits for all the scheduled rules to complete their execution before scheduling the next set of independent rules. This step can make some rules wait *unnecessarily*. For example, consider the rule graph shown in Figure 5.2. It is scheduled in the following sequence of groups $\{r1 \rightarrow (r2, r3) \rightarrow (r4, r5)\}$ and note that the rule r4 waits until the completion of r2 and r3 in spite of the fact that it is eligible for the execution as soon as the rule r2 is completed. To avoid the unnecessary waiting of rules, the above algorithm is modified. The modified algorithm does not wait until the completion of all the scheduled rules. Instead, it selects the next set of independent rules (if any) for scheduling after the completion of every scheduled rule. The wait step monitors the scheduled rules for the completion of at least one of them. The completed rules place their acknowledgments in a queue asynchronously. The wait step waits only when the queue is empty, otherwise proceeds to the `remove` step. In the `remove` step, all the rules corresponding to the acknowledgments in the queue are removed from the rule graph and also the outgoing edges. The `remove` step also removes the acknowledgments from the queue. This approach avoids the unnecessary waiting of rules and thereby increases the concurrency among rules.

5.2.4 Locking Scheme

Although the above scheduling algorithms enforce the control structure, they do not take care of the *isolation* of concurrently executing rules, e.g., concurrent

execution of two independent rules need not be serializable. The following locking rules are used for maintaining the rule isolation as well as the transaction (TG) isolation.

1. *A transaction/rule may hold a lock in write mode (read mode) if all other transactions/rules holding the lock in any mode (in write mode) are ancestors of the requesting transaction/rule. Note that, for a rule, triggering transaction (or rule) becomes the parent and any rule (or transaction) above in the triggering line of hierarchy becomes an ancestor.*

2. *When a transaction/rule aborts, all its locks, both read and write, are released. If any of its ancestors hold the same lock, they continue to do so in the same mode as before the abort.*

3. *When a transaction/rule commits, all its locks, both read and write, are inherited by its parent (if any). This means that the parent holds each of the locks, in the same mode as that of the committed child.*

Theorem 2: *The proposed concurrency control method which comprises of the locking scheme together with the topological scheduling algorithm is correct.*

Proof: *The concurrency control method is correct if and only if the following conditions are satisfied: i) the concurrent execution of rules in a triggered rule graph is equivalent to some topological order of execution, and ii) the concurrent execution of transactions is equivalent to some serial order of execution.*

It is proved in Theorem 1 that the proposed topological scheduling algorithm maintains the topological order for a given rule graph. But, according to the scheduling algorithm, there can be several rules executing concurrently. For example, when

a scheduler identifies several independent rules, all of them are scheduled for execution at the same time. In the asynchronous approach, there can be some new rules being scheduled when there are rules being executed already. The isolation among all these rules is maintained by the above locking scheme. The first locking rule does not allow two rules to share the locks in a conflicting mode, as long as they do not have an ancestor-descendant relationship. The fact that no two rules in a rule graph have an ancestor-descendant relationship (not to confuse with predecessor-successor relationship) ensures that the concurrently executing rules are isolated. The isolation among the top_level transactions is maintained because the rule locks are inherited by the top-level transaction and they are released only when it commits or *aborts so that other top-level transactions cannot see the partial results until its completion.*

The recovery of the graph-based transactions can be done using the standard recovery methods used for NTM [Moss87, Haer87]. The only difference is that, when a set of rules are being undone, they need to follow the *reverse* of the topological order. A detailed discussion on the recovery techniques is out of the scope of this work.

CHAPTER 6
ARCHITECTURE AND EXECUTION MODEL

An object-oriented knowledge base management system (OOKBMS) is very complex and the incorporation of active rules increases the complexity further. The performance of such a complex system cannot satisfy the requirements of many time-constrained applications when it is implemented on a sequential machine. We therefore undertook the implementation of OSAM*.KBMS/P on a high-performance parallel computer. OSAM*.KBMS/P is an OOKBMS which supports OSAM*/P as the underlying knowledge representation model and the graph-based transaction model as the execution model. We design a client/server-based architecture and an asynchronous execution model for the implementation of OSAM*.KBMS/P. In the remainder of this chapter, we describe the architecture and the execution model.

6.1 Client/Server Architecture

6.1.1 Hardware Architecture

The overall architecture of OSAM*.KBMS/P is based on the standard client/server architecture explained in [Rous91]. The hardware architecture is shown in Figure 6.1. Clients C1, C2,..Cn which are residing on Sun workstations are connected to the server by an Ethernet. We use an nCUBE2 parallel computer as the server because of its high-performance and scalable architecture.

The nCUBE2 system architecture can be characterized as a shared-nothing multiprocessor architecture in which each processing node (or simply node) oversees its own private memory space and potentially its own private disk [nCU92a]. Nodes are

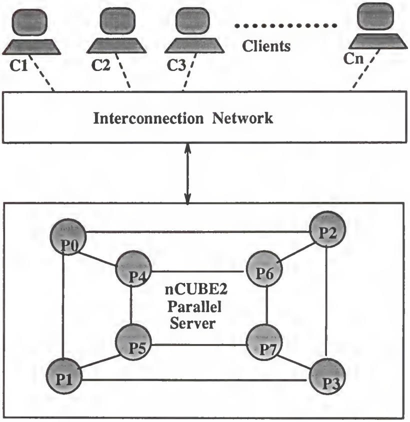
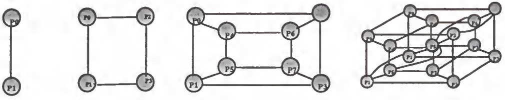


Figure 6.1. Hardware Architecture

interconnected together using a high speed interconnection network. In nCUBE2, each node contains a processing element coupled with a Dynamic Random Access Memory (DRAM) element. The processing element is a single VLSI chip containing a 64-bit CPU, a 64-bit IEEE standard floating point unit, a priority message routing unit which can send or receive messages at a rate of 2.22 Mbytes/sec, and 14 bi-directional communication lines that form a hypercube network with other processing nodes. The processor chip is rated at 7.5 Vax MIPS and 3.3 MFLOPS single precision. One of the 14 bi-directional communication channels of the processor chip is reserved exclusively for the purpose of I/O. This allows each node to function either in a computational mode or in an I/O mode. The other 13 channels are used to

address other computing nodes. This allows for a maximum configuration of 2^{13} or 8192 processing nodes.

The nCUBE2 system architecture is a Multiple Instruction Multiple Data (MIMD) architecture, where each of the nodes can function independently. The backbone of such a system is the interconnection network which connects the nodes together. nCUBE2 uses a hypercube network which can be scaled dynamically to meet the processing needs of a task. Figure 6.2 shows different hypercubes with different dimensions. The speed of the hypercube network allows the OSAM*.KBMS/P transaction server to abstract the nCUBE2 machine as a number of powerful processing elements (some functioning as either computational or I/O processors), interconnected by a very high speed network with a high bandwidth.



Dimension= 1 2 3 4

Figure 6.2. nCUBE2 Architecture

6.1.2 Software Architecture

The software architecture of OSAM*.KBMS/P is shown in Figure 6.3. A client contains an X-Motif graphical user interface (GUI) which has advanced features to browse and edit transaction graphs, rule graphs, OO queries and OO knowledge base schema [Lam92]. Thus, our architecture takes advantage of inexpensive workstations to provide advanced GUI facilities and thereby relieving the server from running and maintaining complex X-Motif GUI software. Clients translate user-submitted transactions into an intermediate form and send them to the server through the

inter-connection network. There is no limit on the number of clients that can be connected to the server. The server gives a unique identifier for each client's request and processes all the requests in a serializable fashion.

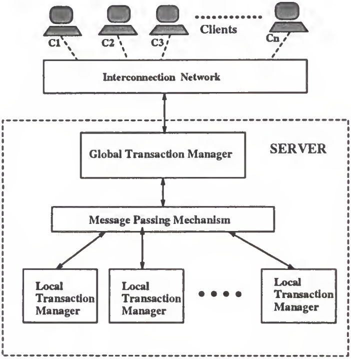


Figure 6.3. The Software Architecture of OSAM*.KBMS/P

As explained earlier, in OSAM*/P, an OO knowledge base schema is defined by a set of classes and their various types of associations. Each class is defined in terms of its structural associations with other classes, a set of method specifications, a set of rule graphs, a set of rules and the implementations of its methods. The knowledge base is partitioned into classes, the instances of each class are stored at a node of the shared-nothing architecture. Multiple classes can be assigned to a single processing node. A global data dictionary which contains the overall schema information and the mapping between classes and the processing nodes is maintained. A local data dictionary, which contains the structure of local classes and rule graphs that can be

triggered by local DB operations (so that operations at a node trigger only the local rule graphs), is stored along with the object classes in a node. It should be noted that rules are also distributed among the classes.

The server is divided into two distinct functional modules – a Global Transaction Server (GTS) and a cluster of Local Transaction Managers (LTMs). GTS handles interactions with clients and oversees the concurrent execution of multiple TGs. Each LTM oversees the concurrent execution of multiple tasks (or rules) assigned to it. All computation-intensive functionalities such as data processing, rule processing, lock management and recovery, are assigned to the LTM module so that the processing burden of the server is distributed among several LTMs. All LTMs residing at several nodes are homogeneous. This enables the server to scaleup well. To enhance the capacity of the system, more processing nodes can be added (maximum limit is 8192). LTMs must be loaded onto the new nodes and the underlying DB must be redistributed accordingly. In this way, system can be scaled up to any extent. However, beyond some size, GTS may become a bottleneck. In that case, GTS can be replicated on multiple nodes and they share the processing of the incoming transactions among themselves. Configurations that allow multiple GTSs are discussed in the next chapter (on implementation).

Each module of the server performs its operations *asynchronously* independent of the other modules and the synchronization is achieved by messages and signals. The detailed asynchronous execution model is explained in the next section.

6.2 Asynchronous Execution Model

As shown in Figure 6.3, the server has a global transaction server (GTS) and a group of local transaction managers (LTMs). GTS receives the TGs coming from different clients asynchronously and supervises their concurrent execution. GTS uses

the asynchronous scheduling algorithm explained in Section 5.2.3 for scheduling TGs and rule graphs, and assigning tasks and rules to underlying LTMs. In addition, it interacts with LTMs to schedule a triggered rule graph at an appropriate time depending on its trigger time. GTS has an efficient message passing mechanism provided by `nread`, `nwrite` commands [nCU92b] to interact with LTMs.

Each LTM receives tasks from GTS asynchronously and processes them in an interleaved fashion. LTM utilizes the GTS's scheduler for scheduling the triggered rule graphs and uses the local lock manager for maintaining the isolation property of rules (tasks). Since database operations generated by a task or a rule are much more complex and time-consuming when compared with the scheduling of GTS, LTMs seldom wait for GTS. The detailed functionalities and algorithms of GTS and LTM are given in the following sections.

The asynchronous behaviour of both GTS and LTM is achieved by having queues which can autonomously receive incoming messages. Each module checks the queue before reading a message from the queue because a "read" command waits until the message is arrived whereas a "check" command returns the status of the queue immediately, without making the module to wait.

6.2.1 Global Transaction Server

The components of GTS are shown in Figure 6.4. GTS consists of a transaction queue (TQ), a transaction scheduler (scheduler), a wait queue (WQ) and an acknowledgment queue (AQ). In Figure 6.4, T1,T2 .. are TGs and G1,G2 .. are acknowledgments from LTMs. GTS keeps the incoming TGs in the rear of TQ. TQ acts as a buffer to offset the mismatch between the rate at which TGs are arriving from clients and the rate at which TGs are being scheduled by the scheduler. The scheduler runs the asynchronous scheduling algorithm explained in Section 5.2.3. It

gets a TG from the head of TQ, selects the independent tasks, distributes them to the underlying LTMs, and keeps the remaining tasks in WQ. At this time, the scheduler, instead of waiting for the scheduled tasks to complete, switches to the next TG in TQ for scheduling. Since there are multiple LTMs each capable of processing a task or a rule, the scheduler is able to schedule TGs in an asynchronous fashion. In addition, each LTM maintains a task queue so that GTS does not have to wait for assigning a task to a busy LTM. For assigning a task to an appropriate LTM, GTS identifies the classes that are used by the task and assigns to LTM that handles the classes. GTS looks into the global data dictionary to find the LTM corresponding to a class. In case, there are multiple classes handled by multiple LTMs, a load balancing strategy is used to select the appropriate LTM.

GTS keeps the TGs that are partially processed and waiting for further processing, in WQ. It can be observed that TGs are being interleaved by the scheduler and also several tasks are being processed in parallel by the underlying LTMs. Also, to schedule a TG asynchronously the scheduler has to be informed when a task completes its execution, because it can make some other tasks in TG eligible for the execution. For this purpose, GTS maintains an Acknowledgment Queue (AQ) which receives acknowledgments from LTMs. Each acknowledgment indicates that an assigned task is completed/aborted. The scheduler periodically monitors AQ to check if any task is completed and schedules the next set of independent tasks. The completed tasks are marked and the TG is kept in WQ. A TG's execution is completed when the scheduler reaches the commit node. GTS requests all relevant LTM's to commit the transaction using 2PC. The committed TG is removed from WQ.

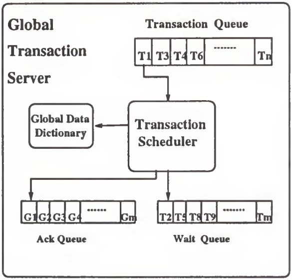


Figure 6.4. Architecture of a Global Transaction Server

The GTS's scheduler, in addition to TGs, schedules rule graphs also. In case of a rule graph with the trigger time *after*, GTS attaches it just before the commit task of the TG as shown in the Figure 5.6.b. The detailed algorithm of GTS is as follows:

```
ALGORITHM GlobalTransactionServer
 LOOP
   IF  Ack_Queue is Empty THEN
      Get transaction T from the Transaction_Queue;
   ELSE
      Get the transaction T corresponding to the acknowledgment
      from the Wait_Queue;
   IF T is a rule graph and the trigger time is "after"
   THEN Attach the rule graph before the commit
        node of the parent transaction;
        Exit the loop;
```

```
ELSE
    IF T is a partially processed transaction
    THEN Mark the completed task;

  Select the independent tasks of T;

  FOR each independent task DO
    Determine the appropriate LTM
    and assign the task.
  ENDFOR
 FOREVER;
END GlobalTransactionServer;
```

6.2.2 Local Transaction Manager

GTS keeps a task in the LTM's task queue for processing. It is LTM's responsibility to execute the tasks in the task queue asynchronously. LTM also maintains a task queue, a wait queue (WQ) and an acknowledgment queue (AQ) to process the incoming tasks and rules in an asynchronous fashion. In addition, an LTM also contains a task processor to process tasks and rules, a data processor to process low level database operations, a lock manager to process lock requests for the local data and a recovery manager to perform local recovery. The LTM architecture is shown in Figure 6.5.

When a task is being processed, an operation generated by the task can trigger a rule graph. Since a rule graph has the same control structure as that of a TG, the task processor utilizes the asynchronous scheduler of GTS for scheduling rules. The

trigger time of the rule graph is maintained by placing it in an appropriate control point of the top_level transaction as explained below.

- In case of *before* as the trigger time, task processor suspends the triggering task and requests the GTS to schedule the triggered rule graph. The task processor may switch to some other task until it receives an acknowledgment from GTS, that the rule graph's execution is completed. The task processor resumes the task as soon as it sees such an acknowledgment in its AQ.

- In case of *immediately-after* as the trigger time, the triggering operation is performed and the task is kept in the local WQ. The task processor then does exactly the same as the case *before*.

- In case of *after* as the trigger time, the rule graph is kept in GTS's WQ and the task processor resumes the processing of the task. GTS appends the rule graph to the top level TG just before the commit task as shown in Figure 5.6.b and eventually it is scheduled just before the top-level transaction is committed.

- In case of *parallel*, the triggered rule graph is kept in the transaction queue and the rule graph is scheduled like a separate TG.

LTM processes the tasks and rules in its task queue in an asynchronous fashion. When the task processor has to wait for some event during the processing of a task, it would keep that task in WQ and mark the control point to resume the processing later and switches to another task. The task processor calls the underlying data processor for performing data processing operations. The serializability for the tasks which are being executed in an asynchronous fashion, is achieved by lock manager (LM). Task processor acquires locks for the data being accessed before sending the

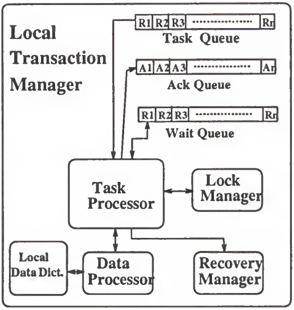


Figure 6.5. Architecture of an Local Transaction Manager

operations to the data processor. An LM receives lock requests for the local data from the local task processor and sometime from remote task processors. The local LM is responsible for the locks on the object class that is stored in that particular node and follows the locking rules explained in Section 5.2.4. The detailed algorithm of the LTM is as follows:

ALGORITHM LTM

LOOP

IF Ack_Queue is Empty THEN

Get Task R from the Task_Queue;

ELSE

Get the Task R corresponding to the acknowledgment

from the Wait_Queue;

ENDIF;

LOOP

```
  Get the next operation for processing and
  Check if operation triggers any rules;
  IF a rule graph is triggered THEN
    CASE trigger_time OF
      before: Mark the operation and keep the
           task in wait queue and place the
           rule graph in the GTS wait queue
           and keep an acknowledgment in the GTS Ack
           queue; Exit;
      immediate after: Obtain locks, perform
           the operation, and keep the
           task in wait queue and place the
           rule graph in the GTS wait queue
           and keep an acknowledgment in the GTS Ack
           queue; Exit;
      after: Keep the triggered rule graph in
           the wait queue and keep an acknowledgment
           in the GTS Ack queue;
      parallel: Keep the triggered rule graph
           in the transaction queue;
  ELSE  Obtain locks for the operation and
        send the operation to the data processor;
UNTIL end of the task;
Send an acknowledgment to the GTS Ack queue to indicate that the
task is completed;
```

FOREVER;

END LTM;

6.2.3 Data Processor

As shown in Figure 6.5, each LTM has a data processor for processing operations on the local database. When a DB operation involves remote data, e.g. the operation is to form a subdatabase which involves multiple classes (analogous to joining of multiple relations), the data processor uses asynchronous multiple wavefront algorithms [Su91, Thak94] to form the subdatabase in parallel. We explain briefly how wavefront algorithms process the OODB operations in a parallel and asynchronous fashion.

As explained earlier, in OSAM*/P, a database is viewed as a *schema graph* intensionally and an *object graph* extensionally. In a schema graph, nodes represent classes and edges represent the associations among the classes (e.g., see Figure 4.1). In an object graph, nodes represent objects and edges represent connections among these objects. For example, the object graph shown in the Figure 6.6.a depicts objects in the classes (RA, Grad etc.) and their inter-connections. The subgraph corresponding to the subdatabase of interest is specified by a Context expression [1], e.g., the Context `RA * Grad * Student * AND (Section,Department)` means: Identify all object instances of RA, Grad, Student, Section and Department which satisfy the query pattern given in Figure 6.6.c, i.e., the sections and departments of graduate students serving as RAs. Following the Context specification, a query would specify the system-defined or user-defined operations which are to be performed on the object instances in the subdatabase. The resulting subdatabase for the above context expression is shown in Figure 6.6.b.

[1]See [Alas89] for full details on the syntax and semantics of the Context expression.

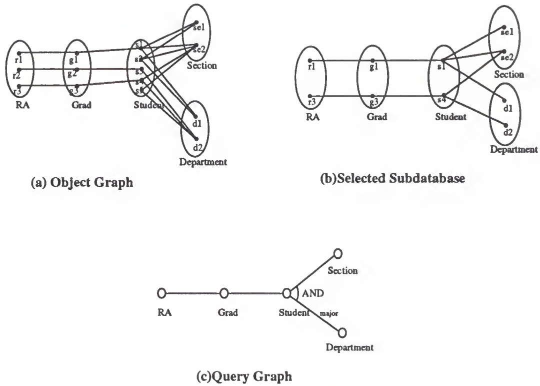


Figure 6.6. Graph-Representation of OODB Operations

Two parallel multiple wavefront algorithms, namely identification [Thak90] and elimination [Su91] algorithms are used for establishing a subdatabase specified by a Context expression. Object classes are assigned to a number of data processors which manage the instances of these classes. The instances of each class, stored in a data processor, are connected to the instances of the neighboring classes of an object graph. All connections using instance identifier (IID) references are bi-directional. Given a query, the query graph which represents the Context expression of the query is sent to all the data processors that manage the involved classes. The identification algorithm would start from all the terminal nodes (i.e., RA, Section and Department in the above example) which simultaneously propagate the IIDs of the neighboring class's instances to their neighbors. The propagation of each terminal node forms a wavefront, thus multiple wavefronts of IIDs go across one

another. Each data processor receiving IIDs would look up its local tables to identify the IIDs of some other neighboring nodes that are connected with the IIDs just received, and propagate the IIDs to its neighbors. A set intersection or a set union operation is performed on the IIDs of multiple waves depending on whether AND or OR branching operator is used in a query graph. When all the terminal nodes have received and processed all the waves of IIDs, the subdatabase would be established and the algorithm terminates.

The elimination algorithm also uses the multiwavefront strategy except that wavefront propagations start from all nodes (instead of only terminal nodes). Each node eliminates instances that do not satisfy the query pattern instead of identifying those that do. The uneliminated instances form the final subdatabase.

Each data processor is capable of running both identification and elimination algorithms which have different performances under different data and query conditions. Data processors select the appropriate algorithm based on these conditions.

6.2.4 Lock Manager and Recovery Manager

In OSAM*.KBMS/P, consistency and correctness of the concurrent execution of tasks, transactions, and rules, are maintained using a distributed Lock Manager. Each LTM residing at a node contains a Local Lock Manager which is responsible for granting/rejecting locks on the data at that particular node. Each Local Lock Manager comprises of a Lock Communication Manager (LCM) and a Lock Logic Manager (LLM). LCM is responsible for dispatching lock requests to the appropriate LLM depending on the location of the data item for which a lock is requested. LLM is responsible for processing lock requests using the locking rules given in Section 5.2.4. The detailed implementation of the distributed Lock Manager is discussed in the next chapter.

Our approach for recovery is as follows. Since our transaction is decomposed into several tasks and the task isolation is maintained, a finer control over recovery is possible. When a task is aborted, it can be recovered without affecting other tasks or transactions. Our approach is similar to that of the nested transaction recovery explained in [Moss87, Roth89]. It is based on write ahead logging and UNDO/REDO strategy which is more efficient than the version approach. However, the recovery of a graph-based transaction is different from that of a nested transaction in that the tasks (or subtransactions) need to be undone in reverse of the order they are scheduled (reverse topological order) in order to maintain the semantic consistency. This approach is efficient because several recovery managers (at different nodes) can undo tasks in parallel. In the current system, the recovery manager has not been implemented.

CHAPTER 7
IMPLEMENTATION ON nCUBE2

OSAM*.KBMS/P is implemented on an nCUBE2 computer [Li93, Cher93, Nart94]. As explained earlier, two main components of the system are GTS and a cluster of LTMs. In this chapter, we first explain the implementation of GTS and then explain the implementation of Task Processor, Lock Manager and Data Processor which are the major modules of an LTM.

7.1 Implementation of Global Transaction Server

GTS is the front-end of the server. It receives transaction graphs (TGs) from clients, translates them into a main memory representation (from a message format) and schedules the tasks in each transaction graph (TG) according to their control structure. GTS maximizes the utilization of nCUBE2 processors by executing several tasks in parallel and interleaving the execution of multiple TGs.

The GTS implementation is partly based on the *Many Servers-Many Schedulers* transaction processor structure described in Gray and Reuter [Gray93]. It consists of two components which can be configured in a number of different ways. The first component is called Transaction Launcher (TL) and is responsible for interacting with different clients. The second component is called a Transaction Scheduler (TSC) and is responsible for enforcing the control structure among tasks. Figure 7.1 shows different ways in which a GTS can be configured. From Figure 7.1, one TL can communicate with many TSCs, and one TSC can communicate with many TLs. Any (or all) of the three configurations shown in Figure 7.1 can be active at any time.

A suitable configuration can be chosen according to the application demands. The functionality of GTS can be replicated by having multiple configurations active and operating independently on different streams of incoming TGs.

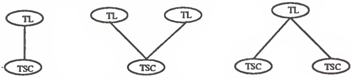


Figure 7.1. Different Configurations of the Global Transaction Server

7.1.1 Transaction Launcher

The main function of the TL is to isolate the server from dealing with different network and user interfaces, and also dealing with the overhead of preparing submitted TGs for execution. Figure 7.2 shows the execution model of a TL. Clients send their transactions to the Interface Handler (IH) which queues them for the Transaction Parser (TP) to read. IH is currently implemented as a spooling application. Clients spool their transaction requests (in the form of messages) to IH which hands them over to TP on a FIFO basis. An analogy is a print spooling application, except that instead of printing, the transaction execution is initiated.

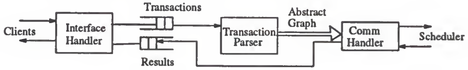


Figure 7.2. Execution Model of a Transaction Launcher

The primary responsibility of TP is to take the high-level specification of a TG, and translate it into an internal memory representation, without loss of information about a TG's execution semantics. A high-level specification of a TG consists of a set of task definitions. Each task has three components namely, a `TASK_NAME` which is unique among all tasks in a TG, a `TASK_BODY` that specifies a sequential list of

queries which makes up a sub-unit of database work, and a CONTROL_ASSOCIATIONS section which contains the control relationships of the task being defined, with the other tasks in TG. An example TG definition corresponding to the "Registration transaction" shown in Figure 7.3, is given below.

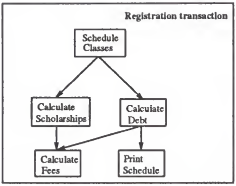


Figure 7.3. Control Diagram of a Graph-based Transaction

BEGIN_TRANSACTION RegistrationTransaction

TASK_NAME: ScheduleClasses;

TASK_BODY: Q1, Q2, .. Qn;

CONTROL_ASSOCIATIONS:

P: CalculateScholarships, CalculateDebt;

TASK_NAME: CalculateDebt;

TASK_BODY: Q3, Q4, .. Qm;

CONTROL_ASSOCIATIONS:

S: PrintSchedule;

TASK_NAME: CalculateScholarships;

TASK_BODY: Q5, Q6, .. Qs;

TASK_NAME: CalculateFees;

TASK_BODY: Q1, Q2 .. Qn;

CONTROL_ASSOCIATIONS:

Y: CalculateScholarships, CalculateDebt;

TASK_NAME: PrintSchedule;

TASK_BODY: Q1, Q2 .. Qn;

COMMIT RegistrationTransaction;

Translating the high-level specification of a TG to its corresponding memory image, is an operation tantamount to capturing the immediate predecessor/successor relationships among tasks into a main memory data structure. A GRAPH data structure is used to capture tasks as nodes (or TASK_BLOCKs) and their relationships as directed edges. The following algorithm is used for translating a TG in the message format into the GRAPH structure.

```
Algorithm TranslateGraph(Transaction graph in a message form)
 struct GRAPH *GraphNode;
 GraphNode := BuildGraphNode(TG_message );
                /* get info about the transaction graph */
 GraphNode.TASK_BLOCK := BuildTaskblocks(TG_message);
                /* get info about tasks into TASK_BLOCKs */
 AddDependencyInfo(GraphNode.taskblock, TG_message);
                /* get info about control relationships */
 return GraphNode;
End TranslateGraph
```

The BuildGraphNode function scans a message (TG) and puts the transaction information (e.g., transaction name, ID, etc.) into the GRAPH data structure. The BuildTaskblocks function puts the task information (e.g., TASK_NAME and TASK_BODY) into TASK_BLOCKs of the graph. AddDependencyInfo adds the control structure information to the TASK_BLOCK. In addition, it fills the indegree (number of predecessors) field of each node. Before scheduling the translated TGs for processing, TRID manager is invoked to assign unique IDs to TGs and all the tasks within.

7.1.2 TRID Manager

Transaction Identifier (TRID) manager is responsible for assigning unique IDs for all TGs and tasks. Soon after the translation of a TG into a memory image, TRID manager is called to assign a TRID to that TG. The TRID is the glue that binds all the actions of a TG together. Within OSAM*.KBMS/P, all TRIDs assigned to currently executing transactions are unique. All processing nodes in a hypercube on nCUBE2 have unique node_id numbers. Globally unique TRIDs are assigned by concatenating the node_id of the node where the TSC module is executing (in case there are multiple TSCs in the configuration), with a sequence number which is guaranteed (by the TRID Manager) to be unique for all transactions executing at a particular node. There are several benefits to assigning a TRID in this manner: (i) it allows each TSC to autonomously assign and manage its own resources, (ii) it takes advantage of the scalability of the shared-nothing architecture, and allows multiple TSC units to come into, and go out of existence with no complications or interference with the other system modules. Once a TRID has been assigned, a TG is passed to a TSC for scheduling.

TRID manager assigns unique IDs to tasks also (application tasks or rules). All TGs contain tasks, and in order to maintain the atomicity and isolation principles of tasks, all tasks within a TG are assigned unique identifiers (TASK_ID). In order to allow other modules to independently determine the lineage of a task, a task's family history is incorporated into its TASK_ID in the following manner. Whenever a subtask is spawned, the new task generates its TASK_ID by appending a number which is unique among all its siblings to the TASK_ID of its ancestor. In general, most root tasks (application tasks that are defined in a top-level TG) are not expected to generate subtasks, therefore as an optimization technique all root TASK_IDs are packaged in one structure with a TRID. Once a root task spawns subtasks (or triggers a rule graph), a linked list format is used to represent the subtasks' TASK_IDs. The root TASK_ID is just one structure, however the subtask's TASK_ID is composed of the root TASK_ID as well as a linked list. It should be noted that from a TASK_ID one can discern where in the system it was created, what LTM is responsible for its execution, and also the TASK_IDs of all its ancestors. Such information allows modules such as the Lock and Recovery Managers to act based on a TASK_ID, without cluttering the communication network with inquiries about tasks.

7.1.3 Transaction Scheduler

The responsibility of a TSC is to schedule the tasks in a TG according to their control structure. The TGs to be scheduled are assigned to it by a TL. It keeps a list of active transactions in the system and "multi-tasks" between them. Basically, whenever a TG is blocked, the scheduler switches its focus to another TG that has pending tasks. Figure 7.4 shows the execution model of a TSC module.

The Scheduler Logic (SL) controls the execution of a TSC module. It is an asynchronous module which acts based on the state of the message queue. When

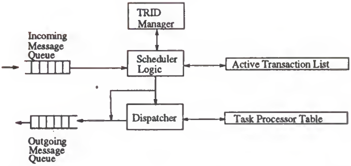


Figure 7.4. Execution Model of a Transaction Scheduler

the queue is empty, the SL module is blocked, otherwise it reads and responds to messages in the queue. The modules that communicate with a TSC module through the message queue, are basically TLs and LTMs. TLs send new incoming TGs to the message queue, and LTMs inform the TSC about the completion of tasks that have been assigned to it.

SL ensures that no task of a TG is scheduled for execution until all its predecessors complete their execution (thereby enforcing the control structure). SL makes use of the control structure information captured by the Transaction Parser (in particular the Indegree and the SuccessorList (adjacency list) fields of the TASK_BLOCK). The Indegree field captures the number of immediate predecessor tasks which have to complete their execution for a particular task to be scheduled for its execution. Whenever the indegree of a particular TASKBLOCK is zero, it semantically means that all predecessors of the corresponding task have completed their execution, as such the task can be scheduled for execution. For example, in the graph shown in Figure 7.3, task `ScheduleClasses` has indegree=0, therefore it is scheduled first. When SL receives acknowledgment from the LTM (by means of the message queue) about the completion of execution of a task, SL traverses the task's SuccessorList

decrementing by one, the indegree of all the task's immediate successors. All immediate successors with an indegree field of zero, can be scheduled immediately for execution. Others with indegree fields greater than zero have predecessors which have not completed execution yet, as such, cannot be scheduled for execution. Once a task completes its execution, it is removed from the graph, when no more tasks remain, a TG completed its execution, and SL broadcasts a Commit message to all LTMs involved in the execution of that TG. We use the 2-phase commit protocol to commit a TG. The algorithm for the SL is as follows.

```
Algorithm SchedulerLogic
 Repeat
  Wait until there are messages in the Message Queue
  While there are Messages in the Queue
    Read Message from Queue
     case Message = NEW_TRANSACTION
         Get a TRID for the new transaction T from TRID Manager
         For each task ti in T
            If ti.indegree = 0
               Dispatch(ti);
               Mark ti as 'In Execution';
            EndIf;
         EndFor;
         Continue;
     case Message = TASK_DONE
         Find the Transaction T that the task ti belongs to;
         Find the task ti in T;
```

```
            Mark ti as 'Completed Execution';
            If ti.SuccessorList exists
               For each member Mi of ti.SuccessorList
                  Get the TASKBLOCK tb located at Mi.Taskblock;
                  decrement tb.indegree by 1;
                  If tb.indegree = 0
                     Dispatch(tb);
                     Mark tb as 'In Execution';
                  EndIf
               EndFor
            Else
               If all tasks in T are marked as Completed Execution
                   Send TRANSACTION_COMMIT message to LTMs
            EndIf;
            Continue;
      EndWhile
   EndRepeat
End SchedulerLogic;
```

The function Dispatch (in the above algorithm) is responsible for sending tasks, to different LTMs. Each LTM has a Task Processor (TProc) to oversee the execution of tasks assigned to it and the rules triggered thereby. The Dispatcher keeps a table for all active TProcs in the system, called the Task Processor Table. Associated with each entry in the table, is the current work load of TProc. The Dispatcher assigns tasks to TProcs, by searching the Task Processor Table to find TProc with the least workload, and assigning the task to that.

7.2 Task Processor

Processing a task in an OOKBMS implemented on nCUBE2 is complex (in terms of computation, communication and control), because (i) for every possible trigger operation, the whole rule base must be searched for detecting triggered rule graphs, (ii) the messages from transaction scheduler, rule processor, lock manager and data processor must be handled asynchronously , and (iii) the control structure among triggered rules must be maintained. It is the task processor's responsibility to process the assigned tasks efficiently. The execution model of a task processor (Tproc) is shown in Figure 7.5. It contains a *Communication Handler* to handle messages from various modules and uses a *Rule Detector* to detect triggered rule graphs efficiently. Rule graphs that can be triggered by local operations are stored in the *local data dictionary.* Rule graphs are stored in an intermediate form so that they need not be parsed every time they are triggered. A TProc also launches rule processors (RProcs) dynamically, to process rules. For instance, to process an assigned rule, TProc launches an RProc as a separate process, and hands over the rule to the RProc for processing.

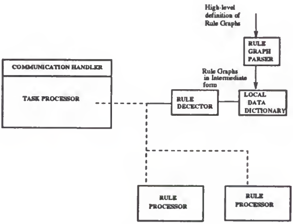


Figure 7.5. Task Processor Execution Model

7.2.1 Communication Handler

A TProc may receive multiple tasks, belonging to different TGs, from GTS. The communication handler (CH) maintains the incoming tasks in a **task queue** from which TProc picks up the tasks. GTS sends several request-messages (e.g., *process_task*, *abort_task*, *commit_task*) to the CH and CH responds back with the appropriate messages (e.g., *task_done*, *task_aborted*, and *task_committed*) when the request is serviced by TProc. The CH also dispatches all lock requests made by the Rule Processor (RProc) and TProc to the local lock manager.

7.2.2 Rule Detector

High-speed detection of the triggered rule graphs is an important requirement to achieve good performance, because the rule detection is part of the critical execution cycle, i.e., it has to be done before every DB operation for detecting the triggered rule graphs. The speed with which it can be carried out largely determines whether an active OOKBMS server is practical or not. In our system, this overhead is reduced by partitioning the rule base and providing a hashing mechanism for searching each partition.

As we mentioned earlier, our rule base is distributed among classes. The Rule Detector (RD) limits its search to one particular class by identifying the class in the trigger operation. For example, for the trigger operation "Update Employee.Salary" RD limits its search to only the Employee class. Rule graphs are distributed in such a way that all the rule graphs that can be triggered by any operation on instances of a class are stored in the definition of that class. This limits the search space of RD to one class for a given trigger operation. For some rule graphs, a trigger operation (especially composite events) may involve multiple operations on different

classes. In this case, the rule graph is replicated in all these classes, whenever one of the operations occurs locally, RD sends messages to find the possible occurrences of other operations at the other classes. In many applications, majority of rules are triggered by primitive events, therefore, the replication does not add much to the storage cost. Within each class, rule graphs are hashed by the trigger operation (e.g., Update Salary, Delete Employee etc.). With the above optimization technique, triggered rule graphs are detected in a constant time, against O(n) of a centralized rule base of n rules. In the above example, using the hash function, RD can directly access the hash bucket which contains all the rule graphs triggered by "update to Employee.Salary".

Our approach causes some overhead during rule base updates (which are rare compared to the rule detection) for incorporating the updates consistently on all replica of rule graphs, and placing rule graphs in appropriate hash bucket and rearranging the hash buckets. However, the fact that the rule detection is in critical execution cycle and rule base updates are very rare implies that the adopted approach is profitable for many applications.

7.2.3 Pretranslation of Rules

Rule graphs are defined as a part of the schema, once defined for an application, they are seldom changed. We exploit this static nature of rule graphs (in contrast to the dynamic nature of data) and translate rule graphs into an intermediate form which is similar to the GRAPH structure of a TG. In addition, for each query expression (in condition, action and otherwise) an equivalent query graph is generated. The pretranslated rule graphs are stored in a Local Data Dictionary and are retrieved when they are triggered. This makes the rule processing faster. When the rule base needs to be modified, the rule designer directly interacts with "rule editor" which

makes the intermediate form of rules transparent and translates all the modifications into the intermediate form automatically.

7.2.4 Rule Processor

A TProc, when rule graphs are triggered, hands over all triggered rule graphs to TSC for scheduling at an appropriate time. TSC schedules rules to different TProcs. For each rule, TProc launches a rule processor (RProc) as a separate process and hands over the rule to it. Each RProc is responsible for processing the assigned rule. As mentioned earlier, each of condition, action and otherwise parts of a rule can be a boolean expression, a query expression (in the query graph form) or an error message. If condition is a query, it is sent to the data processor else it is evaluated by RProc itself. If the condition evaluates to true or returns a non-null data set, then action is executed else otherwise part is executed. If an operation generated by a rule in turn triggers a rule graph, depending on its trigger time, the rule graph is handed over to TSC at an appropriate time for scheduling. The triggering hierarchy is maintained in the IDs of triggered rules. The algorithm adopted by RProc is given below.

```
Algorithm: Process_Rule
Input:  A rule and its id (RULE_ID);
   If Condition is a query
      Process_query(Condition, RULE_ID);
   Else Evaluate Condition;
   If (Condition == TRUE) or (Condition ≠ NULL)
      If Action is a query
           Process_query(Action, RULE_ID);
```

```
    Else Execute Action;
  Else
    If Otherwise is a query
        Process_Query(Otherwise, RULE_ID)
    Else  Execute Otherwise;
  Return result;
End Process_Rule;
```

```
Algorithm: Process_Query
Input : A query graph to be processed by the query processor.
  For each operation in the query graph Do
    Lock_Request(task_id, operation) /* Request locks for the operation */
    /* A task_id can be a RULE_ID of a rule or a TASK_ID of a task */
    Detect_rule_graph(operation)
        /* Check to see if the condition will trigger any rule graph */
    If (triggered_rule_graph > 0)
      If(trigger time = "Before" )
        Process_rule_graph    /* Process the triggered rule graph */
        Send_to_DP;           /*Send operation to Data Processor*/
        break;
      If(trigger time = "Immediate_After" )
        Send_to_DP;
        Process_rule_graph;
        break;
```

```
    If( trigger time = "After" )
        Send_to_DP;
        Process_rule_graph after commit message from GTS;
        break;
    If( trigger time = "Parallel" )
        Send the rule graph to GTS for execution as a separate transaction;
        Send_to_DP;
        break;
  Else
    Send_to_DP ;
 EndFor;
  Return result from query processor;
  Lock_Release(task_id); /* Release the data locks held by the query*/
End Process_Query;
```

7.3 Implementation of Distributed Lock Manager

In a parallel and distributed environment where it is possible for simultaneously executing transactions to concurrently access a mutually shared resource, there has to be an entity which controls the concurrent access of these transactions. Such an entity is called a Concurrency Control Manager. There are various methods of imposing concurrency control (timestamps, static analysis of transactions etc.), when locking is used as a means of imposing concurrency control, the entity that manages locks associated with data items is known as a Lock Manager. In implementing a Lock Manager, it is imperative that the implementation does not add as much overhead to the transaction execution, such that it becomes faster to execute concurrent transactions in a sequential fashion without a Lock Manager. If it becomes faster

to execute transactions in a sequential fashion, then the implementation of the Lock Manager is redundant, and should be removed.

There are two main approaches to implementing a lock manager in a parallel environment. The approach utilized is very much dependent on the hardware and software architecture of the system under implementation. The first approach takes the view that the hardware architecture of the system is a number of multiprocessors which have access to a shared memory. The operating system abstracts this model to provide each process with its own local memory, and access to some shared global memory. The lock information for data items currently being accessed are kept in the shared global memory. Transactions which need locks, coordinate their accesses to the global shared memory, in order to obtain the lock information associated with data items. This method is akin to having a lock manager which allows internal parallelism on its data structures.

The second approach is based on the notion of the non-existence of a shared global memory. In this case, the lock manager has to be implemented as a serial reusable program. Transactions coordinate their access to shared data, by having the serial program process all lock requests. This is analogous to having a queue to which all lock requests are sent, and a lock manager process which dequeues and processes the requests in a serial fashion. However, if there is a single lock manager responsible for locks on all data objects, it is a potential bottleneck.

The nCUBE2 system architecture does not allow processors to share memory, as such there is no such entity as a global shared memory. The nCX operating system which runs on nCUBE2 processors, further abstracts the nCUBE2 hardware architecture, by building a process structure which does not permit processes (even those running on the same processor node) to have access to a shared memory [nCU92a].

Processes can only share information by sending each other messages. The characteristics of nCUBE2's operating environment therefore dictates that the second approach be used in a lock manager implementation. The parallelism is achieved by partitioning all lock requests into groups, and having a local lock manager responsible for each group.

The implementation of the local lock manager is divided into two components, namely a front-end called a Lock Communications Manager (LCM), and a back-end called a Lock Logic Manager (LLM). The LCM acts in a manner similar to a Remote Procedure Call. When an LCM is activated, it finds out which lock group the current request belongs to, and sends the lock request to the LLM responsible for that group. The LLM is the lock process that enforces the concurrency control algorithm. The current software architecture places an LCM and an LLM at every Local Transaction Manager. This does not have to be the case as LCMs and LLMs can be placed at any location in the system.

7.3.1 Lock Communication Manager

The main function of an LCM is to direct all locally generated lock requests to an appropriate LLM depending on the data item to be locked. Every lock request is accompanied by the TASK_ID of the requesting task, the name of the item (Lock_Name) which is to be processed for a lock request, the operation to perform, and the maximum time limit (Time_Out) a requester is willing to wait for a reply from the lock manager. The operation to perform can be any one of the following: READ_LOCK, WRITE_LOCK, TASK_UNLOCK, TRAN_UNLOCK. The lock manager responds with one of the following messages depending on the operation: LOCK_OK, UNLOCK_OK, LOCK_TIMEDOUT. The only interface to a lock manager is through the LCM.

When the LCM is activated, it first finds out which lock group Lock_name belongs to, and packages all its input parameters with the exception of Time_Out into a string format, and sends it to the Lock Logic Manager (LLM) for processing. LCM then goes into a sleep state until it either receives a message from the LLM or Time_Out seconds expire. If LCM receives a message from the LLM, that message is returned by LCM, otherwise a time-out procedure is invoked.

7.3.2 Lock Logic Manager

An LLM is the back-end of a Lock Manager. It is at the LLM that all lock processing is done. The operating environment is such that an LLM receives lock request operations from different LCMs, the LLM processes the lock requests and sends back replies to the LCMs. In cases where a request cannot be processed immediately, the request is queued until lock conditions permit it to be processed.

An LLM uses a lock hash table for accessing information about the status of a lock. In most cases, locks for items are free and hence occupy no space in the system. A lock comes into the existence once an entity requests access to it. The lock header summarizes all the information about the status of a particular lock. The absence of a lock header indicates that a particular lock is free. All lock names which hash to a particular hash value have lock headers which are chained together.

An LLM maintains two queues namely a granted queue and a waiting queue. The granted queue is a list of transactions (and tasks) which have been granted access to the lock. The waiting queue consists of tasks which are waiting for the lock.

The granted queue is actually a queue of trees. At the root of all trees, is a transaction lock. The transaction lock is created when a task of a transaction requests a lock (that is grantable) for an item that has never been locked by the transaction. After a transaction lock has been granted, a tree is built for all the tasks and subtasks

of the transaction which request locks that can be granted on the item. The placement of a task (subtask) in the tree is dependent on its place in the task hierarchy. Information about a task's hierarchy is garnered from its TASKID.

LLM follows the locking rules discussed in Section 5.2.4. The waiting queue contains all tasks and transactions which request locks that are in a conflicting mode with the current mode of the lock. Lock scheduling at the wait queue is based on a slightly modified First In First Out (FIFO) policy. The waiting queue at a lock header is prioritized in the following manner - a task which belongs to a transaction that already has access to the lock, is always queued ahead of a task that belongs to a transaction which has yet to be granted access to the lock. The former wait situation is called a TASK_WAIT, whereas the latter is called a TRANSACTION_WAIT. The queuing policy also places compatible lock requests in the wait queue using the above policy, if there are pending TASK_WAIT requests on the wait queue. This is done to prevent the possibility of livelock.

7.3.3 Deadlock Management

Concurrently executing tasks (or transactions) compete for locks during their execution lifetime. When a task requests a lock that is in conflict with the current mode of the lock, the task goes into a wait state. A task that is in a wait state, stays in that state until the task that is holding the lock in the conflicting mode completes its execution and releases its hold on the lock. In such a system, it is possible for currently executing tasks to start waiting indefinitely for each other in a manner which forms a cycle. Such a wait situation, is termed in the literature as deadlock.

There are three main ways of handling deadlocks: they are (i) developing strategies for preventing deadlocks from occurring, (ii) allowing deadlocks to form, then

detecting and breaking the deadlocks, (iii) avoiding deadlocks by allowing transactions to wait for a specified period of time, and then aborting and restarting them. The first method is not a viable alternative because it involves having all transactions pre-declaring the resources that they require before they can be executed. The second option is in general not a practical alternative, because of the extra overhead and processing that deadlock detection adds to the lock manager implementation. The OSAM*.KBMS/P's lock manager implementation takes a modified version of the third approach known as deadlock avoidance. In particular, the Lock Manager does not abort transactions/tasks. It dequeues a lock request (in the hope of breaking a possible deadlock), and gives the lock requester the choice of either resubmitting the lock request or aborting the entire transaction/task. A call to the LCM requires a parameter which reflects the length of time a lock requester is willing to wait. When that time expires and the LLM has not replied, LCM invokes a time-out procedure which communicates with the LLM using the protocol specified below.

```
Algorithm LCM_TIMEOUT (Lock_Request)
  /* purpose: The LCM_TIMEOUT is invoked when a lock requester has been */
  /*          waiting too long for a reply from an LLM.                 */

    Query the LLM about Status of the lock request;
    Return lock request status to user when LLM replies;
End LCM_TIMEOUT

Algorithm LLM_TIMEOUT(Query)
  /* purpose: Finds out the status of the lock request which caused  */
```

```
/*        the Timeout procedure to be invoked.  If the lock        */
/*        request is in the queue LLM dequeues it, breaking a      */
/*        possible deadlock; otherwise it assumes that its         */
/*        LOCK_GRANTED message  and the Query message crossed      */
/*        paths along nCUBE2's message path, and sends no response*/

    Search the waiting queue for entry associated with the request of the Query;
    If the request is found
        Dequeue request from Queue;
        Send LOCK_TIMEDOUT message to LCM
End LLM_TIMEOUT
```

Since it is possible for an LLM's reply to be in transit when an LCM invokes the timeout procedure, the LLM doesn't respond to TIMEOUT query messages if it doesn't find the lock request for a query message in its wait queue. It is worth mentioning that the timeout procedure does not preclude using deadlock detection procedures. By setting the waiting time of task lock requests to large values, it is possible to perform deadlock detection before the timeout period expires. Examples of various distributed deadlock detection schemes can be found in [Elma86].

7.3.4 Implementation of Data Processor

A data processor is capable of running both identification and elimination algorithms explained in the previous chapter. Both algorithms use a general data structure which stores the object graph information in such a way that it can be efficiently used by both algorithms. The connections between the object instances belonging to two classes are represented by an adjacency matrix. Rows and columns

of the matrix are stored as adjacency lists of IIDs in the data processors that manage those two classes. The adjacency list for each instance contains IIDs of the connected objects in the associated class. For supporting both algorithms, a set of adjacency lists are stored in a data structure that uses *backward pointers.* Instead of storing connected IIDs of each instance in the class, we store the list of local IIDs that are connected to each instance of a neighbouring class. This data structure is useful for both identifying the connected objects and eliminating the objects that are not connected.

Each data processor is implemented as a finite state automata controller (FSAC), which enables the interleaved processing of multiple queries concurrently. When a query sends its wavefront to a neighboring class and waits for the response, the data processor suspends the query by saving its state and switches to another (new or old) query. It moves from one state to another depending on the message received. For example, FSAC resumes a suspended query (by loading the query's state) only when it receives the message the query is waiting for, e.g., a wavefront from a neighbouring class. Each message has message_type, query_id, destination, source, information. Message_type indicates whether the query is new or partially processed. If it is new, a fresh query block which contains the information of that query is created, otherwise, the current state of the suspended query is loaded from the query block and the processing is resumed. The query_id is used to identify the query. The query blocks are hashed on the query_id. Query information contains the message contents, e.g., a set of IIDs. FSAC selects a method to perform the appropriate action from a FSA table in which, for a given message type and FSA state, the appropriate method to be invoked is stored.

7.4 Performance Evaluation

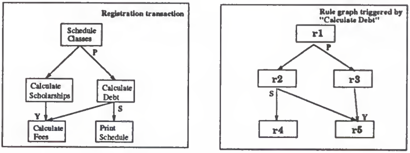
There are two performance metrics in use for parallel systems. The first metric called *speedup*, measures the ability of the system to grow in order to reduce the execution time of a fixed number of tasks. Speedup assesses how effective a system is in allocating its resources. Speedup can be measured using the following equation by keeping the problem size constant and increasing the system size:

$$Speedup = \frac{SmallSystemElapsedTime}{BigSystemElapsedTime}$$

The second metric called *scaleup*, measures the ability of a system to be scaled up to a larger system. In quantitative terms, an n-times larger system is expected to perform an n-times larger job in the same amount of time that it took the original (and smaller) system to process the smaller task. Scaleup can be measured using the following equation by increasing both problem and system in the same proportion.

$$Scaleup = \frac{SmallSystemTimeonSmallProblem}{BigSystemTimeonBigProblem}$$

In an ideal system, as the size of the system is increased, the speedup should increase while the scaleup remains constant. For performance evaluation, we use a "Registration transaction" which has a graph structure shown in Figure 7.6.a, and a rule graph shown in Figure 7.6.b. The latter is triggered by the task "Calculate Debt".



(a) (b)

Figure 7.6. Transaction and Rule Graphs Used for Performance Evaluation

We run a large number of Registration transactions on the server with one global unit (a GTS launched on a cube of 4 nodes) and one local unit (an LTM launched on a cube of 4 nodes). Later, by keeping the number of transactions and global units constant, we gradually increase the number of local units to 15 and note the *speedups* due to the parallel execution properties of the rule and transaction graphs as well as the asynchronous execution model. The speedup is defined as follows:

$$Speedup = \frac{Time\_taken\_by\_640\_transactions\_in\_a\_system\_with\_one\_global\_unit\_and\_one\_local\_unit}{Time\_taken\_by\_640\_transactions\_in\_a\_system\_with\_one\_global\_unit\_and\_n\_local\_units}$$

It can be observed from the curve shown in Figure 7.7 that the increase in speedup is linear up to 7 local units and it gradually tapers as the number of local units reaches 15. The gradual reduction in the speedup is due to the communication time for distributing tasks and rules to different LTMs and the time for converting each of them into a message format suitable for nCUBE2's inter-processor communication. However, it can be observed from the speedup curve that the speedup can be increased further, although not linearly, by increasing the number of local units. The optimal number of local units is 7, to achieve maximum speedup for 640 transactions.

We also evaluate the *scaleup* of the system by running 40 Registration transactions on the system with one global unit and one local unit, and increasing the number of transactions and the number of local units in the same proportion (e.g., 80 transactions on (1 global unit and 2 local units), 120 transactions on (1 global unit and 3 local units), etc.). Formula used for the scaleup is as follows:

$$Scaleup = \frac{Time\_taken\_for\_processing\_1*40\_transactions\_on\_1\_local\_unit}{Time\_taken\_for\_processing\_n*40\_transactions\_on\_n\_local\_units}$$

It can be observed from the scaleup curve in Figure 7.8 that the system has scaled up well because most of the computation-intensive functionalities are distributed to LTMs instead of being centralized at GTS. The scaleup is almost linear up to 480 transactions. At this point, GTS becomes the bottleneck serving multiple LTMs at one side and several clients at another side. The scaleup therefore gradually tapers

as the number of transactions and the number of local units increase further. The scaleup can be further improved by launching multiple GTSs to receive incoming transactions in parallel and schedule them to the underlying LTMs.

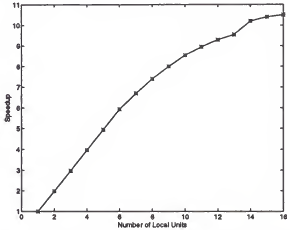


Figure 7.7. Speedup of the Transaction Server

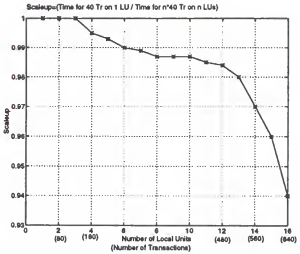


Figure 7.8. Scaleup of the Transaction Server

From the above performance evaluations it can be observed that, because of GTS's configurability and nCUBE2's inherent scalability and high-speed (scalable) communication channels, the speedup and the scaleup do not saturate until a large number of transactions.

We also analyze the execution time of a given rule graph with the increase in the number of local units (LUs). The rule graph shown in Figure 7.9 is executed with different number of LUs. The variation of the execution time as the number of LUs increases is shown in Figure 7.10. Up to 3 LUs the execution time reduces almost linearly and then it gets saturated, in fact, it slowly increases. This is because, according to the control structure of the rule graph, at most three rules can be executed in parallel except in the last phase. Therefore, only three LUs can be utilized well and any additional LU creates an overhead. That is the reason why the execution time increases as the number of LUs increases to more than 3. When 4 LUs are allocated, at least one LU will be idle until the last phase, then R9, R10, R11, R12 can be executed in parallel. However, performance results show that the overhead generated by the 4th LU is more than the speedup achieved.

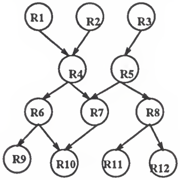


Figure 7.9. A Sample Rule Graph

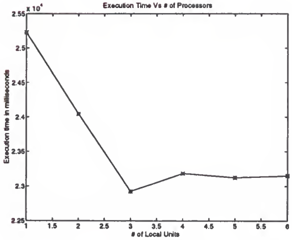


Figure 7.10. Execution Time Vs Number of Processors

CHAPTER 8
SUMMARY, CONCLUSION AND FUTURE WORK

In this research work, we have presented a graph-based rule and transaction processing mechanism for an active object-oriented knowledge base management system and discussed the architecture, algorithms and techniques for implementing it on an nCUBE2 computer – a shared-nothing parallel computer.

The majority of contemporary active DBMSs have adopted priority-based approaches for specifying and enforcing control structures among rules. However, priority-based approaches lack flexibility and expressiveness needed by rules in many advanced database applications. In this thesis, we have introduced a flexible and expressive rule control mechanism based on *rule graphs.* In our approach, we have used graph-based structures (or rule graphs) to capture complex control structures among rules. In our rule specification, the event part of a rule is associated with rule graphs so that rules have the flexibility to follow different control structures when they are triggered by different events. Rule graphs enable the specification of "rule control" at a higher level using CA rules as building blocks. The clear separation of the rule control from rules makes the rule base more understandable and rule controls easier to modify. Also, new rules with desired control requirements can be easily added. Rules can be added in the RULES section and their control requirements can be specified in the RULE GRAPHS section. In addition, we have provided a set of control associations for the specification of rule graphs, using which the parallelism among rules can be explicitly specified.

Since our objective is to incorporate the above rule control mechanism into an object-oriented knowledge base management system, we have uniformly extended an OO knowledge model to capture rule controls in addition to rules. We have introduced control associations to model control relationships among rule objects. This is analogous to modeling semantic relationships among data objects using semantic associations. We have also described an extended rule language using which a rule graph can be specified as a set of control associations. Since the proposed extension is based on the general OO concepts of modeling all entities (including rules) as objects and modeling all semantic relationships among objects as semantic associations, it can be incorporated easily in all the active OODBMSs that support these concepts.

We have used an expressive graph-based transaction model to incorporate the execution of rules with graph-based control structures. In this model, a graph-based transaction (or a transaction graph) dynamically expands its structure to uniformly incorporate rule graphs in the transaction framework. In the transaction graph's structure, there are well-defined control points at which triggered rule graphs are placed according to their trigger times. The graph structure is also general enough to support rule graphs with various trigger times, i.e., rule graphs that are defined to execute before or after any task in the triggering transaction graph, and these rule graphs can be placed at the corresponding control points in the transaction graph. This kind of placement cannot be expressed by linear or tree control structures. On a different note, the transaction graph model can also be used in multi-database environments to specify and enforce inter-task and inter-transaction dependencies.

In addition, in this thesis, we have described the parallel implementation of the OOKBMS that supports the proposed rule model, transaction model, and knowledge representation model. The overall architecture is based on the standard client/server architecture which makes use of inexpensive workstations for GUI processing. The

server which has been implemented on a shared-nothing parallel computer consists of a global component and a cluster of homogeneous local components. All the computation-intensive functionalities are assigned to the local component so that the burden on the server is distributed among several processing nodes. This also enabled the server to be scaled up well. We have also described an asynchronous execution model in which all the transactions and rules are executed in an interleaved fashion. This enabled the server to achieve good speedups. The proposed parallel execution and implementation strategies for rules and transactions are useful for the design and implementation of a parallel production system or a parallel active database system on a shared-nothing computer.

Our future work is to implement the recovery subsystem and enhance the system's rule language with powerful event expressions.

REFERENCES

[Agra91] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 479–487, Barcelona (Catalonia, Spain), September 1991.

[Alas89] A. M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 433–442, Amsterdam, The Netherlands, August 1989.

[Anwa93] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 99–108, Washington, DC, May 1993.

[Atti92] P. Attie, M. Singh, M. Rusinkiewicz, and A. Sheth. Specifying and enforcing intertask dependencies. Technical Report Carnot-245-92, Microelectronics and Computer Technology Corporation, Austin, TX, 1992.

[Bane87] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[Bato85] D. Batory and W. Kim. Modelling concepts for VLSI CAD objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[Beer91] C. Beeri and T. Milo. A model for active object oriented database. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 337–349, Barcelona, September 1991.

[Buch91] A. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed object systems. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 123–158. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Care91] M. J. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Trans. Knowledge Data Eng.*, 3(3):320–336, September 1991.

[Ceri92] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 339–352, Vancouver, September 1992.

[Chak89] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, August 1989.

[Chak94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *to appear in Proc. of 20th Int'l Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.

[Cher93] P. V. Cherukuri. A task manager for parallel rule execution in multiprocessor environments. Master's thesis, Department of Electrical Engineering, University of Florida, 1993.

[Chry91] P. K. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 349–398. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Corm90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 536–538. McGraw Hill Book Company, New York, 1990.

[Daya88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database management system. In *Proceedings 2nd International Workshop on Object-Oriented Database Systems*, pages 116–128, Bad Muenster am Stein, Ebernburg, West Germany, Sept. 1988.

[Diaz91] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 317–326, Barcelona, September 1991.

[Elma86] E. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Record*, 15(3):72–94, September 1986.

[Forg81] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[Forg82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Gatz92] S. Gatziu and K. R. Dittrich. SAMOS: an active, object-oriented database system. *in IEEE Quarterly Bulletin on Data Engineering*, 15(1):23–26, December 1992.

[Geha91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona (Catalonia, Spain), Sep. 1991.

[Geha92] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 81–90, San Diego, CA, June 1992.

[Gray93] J. Gray and A. Reuter. *Transaction processing: Concepts and technques.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[Gupt87] A. Gupta. *Parallelism in production systems.* Pitman Publishing, London, 1987.

[Haer83] T. Haerder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, Dec 1983.

[Haer87] T. Haerder and K. Rothermal. Concepts for transaction recovery in nested transactions. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 12(2):239–247, July 1987.

[Hamm81] M. Hammer and D. McLeod. Database description with SDM: A semantic association model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[Hans89] E. N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *SIGMOD Record*, 18(3):12–19, September 1989.

[Hans93] Eric N. Hanson and Jennifer Widom. Rule processing in active database systems. *International Journal of Expert Systems*, 6(1):83–119, 1993.

[Heil91] S. Heiler, S. Haradhvala, S. Zdonik, B. Blaustein, and A. Rosenthal. A flexible framework for transaction management in engineering environments. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 87–122. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Hsu88] M. Hsu, R. Ladin, and D. McCarthy. An execution model for active database management systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 171–179, Washington, DC, June 1988.

[Huds89] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[Hull87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[Ishi91] T. Ishida. Parallel rule firing in production systems. *IEEE Trans. Knowledge Data Eng.*, 3(1):11–17, March 1991.

[Kuo91] C.-M. Kuo, D. P. Miranker, and J. C. Browne. On the performance of the CREL system. *Journal on Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[Lam92] H. Lam and S.Y.W. Su. GTOOLS: An active graphical user interface toolset for an object-oriented KBMS. *International Journal of Computer System Science and Engineering*, 7(2):69–85, April 1992.

[Lam94] H. Lam, S.Y.W Su, S.R. Eddula, A. Dankar, and S. V. Kunisetty. Model extensibility in an extensible knowledge base management system. *submitted to IEEE Trans. on Knowledge and Data Engineering*, 1994.

[Li93] Q. Li. Design and Implementation of a Parallel Object-Oriented Query Processor for OSAM*.KBMS/P. Master's thesis, Department of Electrical Engineering, University of Florida, 1993.

[McCa89] D. R. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 142–151, Portland, Oregon, June 1989.

[Mira87] D. P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of AAAI 87 Conference on Artificial Intelligence*, pages 42–47, San Diego, CA, August 1987.

[Moss85] E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT, 1985.

[Moss87] E. Moss. Log-based recovery for nested transactions. In *Proc. 13th Int'l Conf. on Very Large Data Bases*, pages 427–432, Brighton, England, September 1987.

[Nart94] R. Nartey. The design and implementatation of a global transaction server and a lock manager for a parallel knowledge base management system. Master's thesis, Department of Electrical Engineering, University of Florida, 1994.

[nCU92a] nCUBE. *nCUBE 2 Processor Manual*, 1992. nCUBE Corporation, Foster City, CA.

[nCU92b] nCUBE. *nCUBE 2 Programmer's Guide*, 1992. nCUBE Corporation, Foster City, CA.

[Rasc91] L. Raschid, T. Sellis, and A. Delis. On the concurrent execution of production rules in a database implementation. Technical Report UMIACS-TR-91-125, University of Maryland, College park, MD, September 1991.

[Rasc94] L. Raschid, T. Sellis, and A. Delis. A simulation-based study on the concurrent execution of rules in a database environment. *Journal on Parallel and Distributed Computing*, 20(1):20–42, Jan 1994.

[Roth89] K. Rothermal and C. Mohan. ARIES/NT: A recovery method based on write ahead logging for nested transactions. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 337–346, Amsterdam, The Netherlands, August 1989.

[Rous91] N. Roussopoulos and A. Delis. Modern client-server DBMS architectures. *SIGMOD Record*, 20(3):52–61, September 1991.

[Schm90] J. G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 65–71, Bostron, MA, July 1990.

[Schm91] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal on Parallel and Distributed Computing*, 13(4):348–365, December 1991.

[Shet91] A. P. Sheth, M. Rusinkiewicz, and G. Karabatis. Using polytransactions to manage interdependent data. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 555–582. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Stol84] S. J. Stolfo. Five parallel algorithms for production system execution on the DADO machine. In *Proceedings National Conf. on Artificial Intelligence*, pages 300–307, Austin, TX, 1984.

[Stol91] S. J. Stolfo, O. Wolfson, P. K. Chan, H. M. Dewan, L. Woodbury, J. S. Glazier, and D. A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal on Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[Ston88] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[Su89] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model (OSAM*). In S. Kumara, A. L. Soyster, and R. L. Kashyap, editors, *Artificial intelligence: Manufacturing theory and practice*, pages 463–494. Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989.

[Su91] S. Y. W. Su, Y.-H. Chen, and H. Lam. Multiple wavefront algorithms for pattern-based processing of object-oriented databases. In *Proc. 1st Int'l Conf. on Parallel and Distributed Inf. Syst.*, pages 46–55, Miami, FL, December 1991.

[Su94] S.Y.W. Su, R.S. Jawadi, P.V. Cherukuri, Q. Li, and R. Nartey. OSAM*.KBMS/P: A parallel, active, object-oriented knowledge base server. Technical report, Department of Computer and Information Sciences, University of Florida, Gainesville, April 1994.

[Thak90] A. K. Thakore, S. Y. W. Su, H. Lam, and D. G. Shea. Asynchronous parallel processing of object bases using multiple wavefronts. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 127–135, St. Charles, IL, August 1990.

[Thak94] A.K. Thakore, S.Y.W. Su, and H. Lam. Algorithms for asynchronous parallel processing of object-oriented databases. *to appear in IEEE Trans. Knowledge Data Eng.*, October 1994.

[Wach91] H. Wachter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 219–264. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Wido91] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 275–285, Barcelona (Catalonia, Spain), September 1991.

[Zert90] D. R. Zertuche and A. P. Buchmann. Execution models for active database systems: A comparison. Technical Report TM-0238-01-90-165, GTE Laboratories, Waltham, MA, January 1990.

APPENDIX
BNF GRAMMAR FOR RULE GRAPHS

```
rule_graph_section : 'RULE GRAPHS' ':' rule_graph_definitions

rule_graph_definitions : rule_graph_definition |
rule_graph_definitions rule_graph_definition ;

rule_graph_definition :  rule_graph_spec |
error   ;

rule_graph_spec : 'rule graph' qualified_name optional_is
rule_trigger
rule_graph_body ';'

qualified_name : <identifier> ;

rule_trigger : 'triggered' trigger_conds;

trigger_conds : trigger_cond |
trigger_conds trigger_cond |
error  |
trigger_conds error ;

trigger_cond : trigger_time operation_list

trigger_time : 'Before' | 'After' | 'Immediately_after' | 'Parallel'

operation_list : operation_spec |
operation_list ',' operation_spec;

operation_spec :  qualified_name '(' optional_param_decs ')' |
OPERATOR qualified_oper_name '(' optional_param_decs ')'

rule_graph_body : rule_definitions
                         'end' optional_ident ;

rule_definitions: rule_definition |
rule_definitions rule_definition ;

rule_definition : rule_spec |
error   ;

rule_spec : 'rule' qualified_name optional_is
                        optional_rule_body
optional_control_associations ';'
```

```
optional_rule_body: <epsilon> | rule_body;
rule_body: optional_rule_cond
optional_rule_action
optional_rule_alt
'end' optional_ident

optional_rule_cond : <epsilon> | rule_condition_spec;

rule_condition_spec : 'condition' rule_condition  |
'condition' error  |
error 'condition' rule_condition  |
error 'condition' error  ;

rule_condition :  expression |
expression '|' expression |
'(' rule_condition ')' ;

optional_rule_action : <epsilon> | rule_action ;

rule_action : 'action' statements |
error rule_action {};

optional_rule_alt : <epsilon> | rule_alt;

rule_alt: 'otherwise' statements

optional_control_associations: <epsilon> | control_associations ;

control_associations : optional_S optional_P optional_Y

optional_S: <epsilon> | s_assoc

s_assoc : 'S:' rule_ident ;

optional_P: <epsilon> | p_assoc

p_assoc : 'P:' rule_idents ;

rule_idents: rule_ident | rule_ident ',' rule_idents

rule_ident: <identifier> ;

optional_Y: <epsilon> | y_assoc

y_assoc :  'Y:' rule_idents ;
```

BIOGRAPHICAL SKETCH

Ramamohanrao Sri Jawadi was born in 1964, in India. He received a Bachelor of Technology degree in Computer Science from the Kakatiya University (Regional Engineering College, Warangal), in October 1987. He worked as a teaching assistant at REC, Warangal, until July 1988. He joined the Indian Institute of Technology, Madras, in August 1988 and was awarded a degree of Master of Science in computer science and engineering in December 1990. After graduation from IIT, Madras, he worked in Wipro Systems Limited, India, until July 1991. He has been a research assistant in the Database Systems Research and Development Center at the University of Florida since August, 1991. He will receive the degree of Doctor of Philosophy in computer and information sciences in December 1994.